Lothar Englisch

Das Maschinensprache Buch

für Fortgeschrittene zum C64 & PC128



EIN DATA BECKER BUCH

DAS STEHT DRIN:
In diesem Buch wird von der Problemanalyse bis zum Maschinensprachealgorithmus in die Grundlagen der professionellen Maschinenspracheprogrammierung eingeführt. Interessant und wichtig für diejenigen, die wirklich alle Möglichkeiten ihres C-64 und PC-128 ausschöpfen wollen.

Aus dem Inhalt:

- Zahlendarstellung
- Die Rechenroutinen des BASIC-Interpreters
- Fließkommafunktionen des BASIC-Interpreters
- Interruptprogrammierung
- Die Benutzung der Timer
- Betriebssystem- und BASIC-Erweiterungen
- Strukturierte Programmierung
- Verwendung neuer Schlüsselworte
- Die Vektoren des Betriebssystems
- Druckerspooling

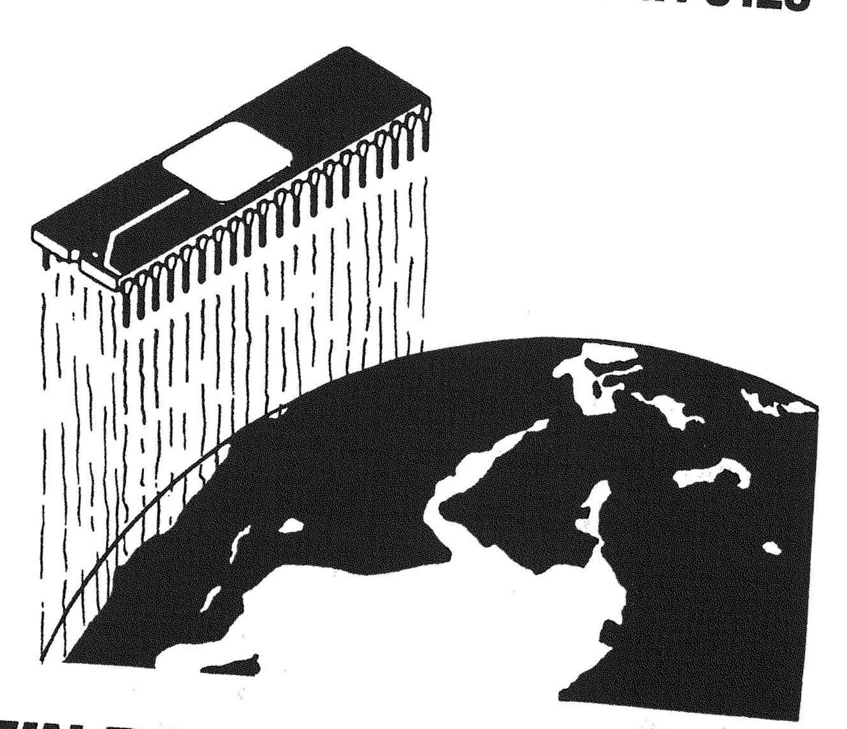
UND GESCHRIEBEN HAT DIESES BUCH:
Lothar Englisch ist Systemprogrammierer in der DATA BECKER Entwicklungsabteilung. Vor allem aber ist er Bestsellerautor (DAS MASCHINENSPRACHBUCH, 64 TIPS & TRICKS, 64 INTERN) und seine Bücher sind anerkannte Standardwerke.

ISBN 3-89011-022-3

Lothar Englisch

Das Maschinensprache Buch

für Fortgeschrittene zum C64 & PC128



EIN DATA BECKER BUCH

ISBN 3-89011-022-3

2. erweiterte und überarbeitete Auflage

Wichtiger Hinweis!

Die in diesem Buch wiedergegebenen Schaltungen, Verfahren und Programme werden ohne Rücksicht auf die Patentlage mitgeteilt. Sie sind ausschließlich für Amateur- und Lehrzwecke bestimmt und dürfen nicht gewerblich genutzt werden.

Alle Schaltungen, technische Angaben und Programme in diesem Buch wurden von den Autoren mit größter Sorgfalt erarbeitet bzw. zusammengestellt und unter Einschaltung wirksamer Kontrollmaßnahmen reproduziert. Trotzdem sind Fehler nicht ganz auszuschließen. DATA BECKER sieht sich deshalb gezwungen, darauf hinzuweisen, daß weder eine Garantie noch die juristische Verantwortung oder irgendeine Haftung für Folgen, die auf fehlerhafte Angaben zurückgehen, übernommen werden kann. Für die Mitteilung eventueller Fehler ist der Autor jederzeit dankbar.

Inhaltsverzeichnis

0 Einleitung

Nachdem in "Das Maschinensprache Buch zum Commodore 64" die Grundlagen der Maschinenprogrammierung beschrieben wurden, liegt nun ein Buch vor Ihnen, das auf diesen Kenntnissen aufbaut und auf die besonderen Möglichkeiten und Fähigkeiten des Commodore 64 eingeht und versucht, mit Hilfe der Maschinensprache alle Möglichkeiten Ihres Rechners zu nutzen.

Besitzer des neuen Commodore 128 können die Beispiele im 64er-Modus direkt übernehmen und ihren Nutzen daraus ziehen. Da BASIC und Betriebssystem im 128er-Modus eigentlich nur Erweiterungen des 64er Betriebssystems sind, dürfte das Übertragen der Programme in den 128er-Modus nicht schwer fallen. Die Tabelle im Anhang sowie das Buch '128 intern' können Ihnen dabei behilflich sein. Im Text wird an entsprechenden Stellen auf Gemeinsamkeiten und Unterschiede zum 128er hingewiesen.

Das Buch ist in drei große Kapitel aufgeteilt, die das Hauptthema dieses Buches sind. Das erste Kapitel über die Zahlendarstellung auf dem Commodore 64 beschreibt ausführlich, wie Ihr Rechner rechnet und wie man seine Arithmetikroutinen auch von Maschinensprache nutzen kann. Neben der Übergabe und Umwandlung von Zahlen in den unterschiedlichen Formaten liegt der Schwerpunkt dieses Kapitels im Schreiben eigener arithmetischer Funktionen, die über die USR-Funktion angesprochen werden können.

Das zweite Kapitel beschäftigt sich mit einem Thema, das eine Domäne des Maschinenprogrammierers ist, der Interrupttechnik. Nach einer Begriffsklärung werden Systeminterrupt sowie die Möglichkeiten zur Auslösung eines Interrupts dargestellt. Durch viele Beispielprogramme wird gezeigt, welche Möglich-

keiten sich hier dem Programmierer bieten. Zum Abschluß dieses Kapitels wird ein Maschinenprogramm vorgestellt, das es ermöglicht, auch in BASIC Unterprogramme interruptgesteuert ablaufen zu lassen.

Das dritte große Kapitel stellt das Konzept der Vektoren in BASIC-Interpreter und Betriebssystem vor. Die einzelnen Vektoren werden beschrieben und mit Beispielprogrammen wird das Einbinden eigener Befehle erläutert. Als Beispiel mag die Implementierung der REPEAT-UNTIL Struktur dienen.

1.1 Zahlendarstellung auf Commodore 64 & 128

Commodore 64 und 128 kennen zwei Arten der internen Zahlendarstellung:

Die erste Darstellung ist Ihnen bereits bekannt und wird bei Variablen vom Typ INTEGER benutzt. Diese Variablen können nur ganzzahlige Werte von - 32768 bis + 32767 erhalten und können mit zwei Bytes dargestellt werden. Von diesen 16 Bits dient das oberste Bit zur Darstellung des Vorzeichens.

dezimal	binär	hex
-32768	1 000 0000 0000 0000	80 00
-32767	1 000 0000 0000 0001	80 01
-32766	1 000 0000 0000 0010	80 02
-32765	1 000 0000 0000 0011	80 03
	...	
-2	1 111 1111 1111 1110	FF FE
-1	1 111 1111 1111 1111	FF FF
0	0 000 0000 0000 0000	00 00
1	0 000 0000 0000 0001	00 01
2	0 000 0000 0000 0010	00 02
	...	
32766	0 111 1111 1111 1110	7F FE
32767	0 111 1111 1111 1111	7F FF

Wir haben es hier also mit vorzeichenbehafteten 16-Bit-Zahlen zu tun, ähnlich wie wir es schon von den entsprechenden 8-Bit Zahlen her kennen, die Werte von -128 bis +127 darstellen konnten und z.B. bei der relativen Adressierung Anwendung finden.

Diese Ganzzahlen mit begrenztem Wertebereich sind jedoch für allgemeine Berechnungen nicht geeignet, da in vielen Fällen

auch Nachkommastellen sowie ein größerer Wertebereich benötigt werden. Um diesen Forderungen gerecht zu werden, hat man die Fließkommazahlen eingeführt. Diese Darstellungsweise kennen wir z.B. auch vom Taschenrechner mit Exponentialdarstellung. Sehen wir uns das Prinzip einmal näher an.

Da wir normalerweise im Dezimalsystem rechnen, fangen wir damit an. Wollen wir eine Zahl darstellen, so schaut man nach, wie oft die Basis des Zahlensystems, also die Zehn, in der Zahl als Faktor enthalten ist und zerlegt die Zahl in zwei Anteile. Ein Beispiel soll dies verdeutlichen:

$$15 = 1.5 * 10^1$$
$$230 = 2.3 * 10^2$$

Wenn wir die Potenzdarstellung auch auf negative Exponenten ausdehnen, so können wir sämtliche Zahlen so darstellen:

$$5 = 5 * 10^0$$
$$0.7 = 7 * 10^{-1}$$

Da die Basis des Zahlensystems bekannt ist, ist eine Zahl durch die sogenannte Mantisse, das ist z.B. die 7 in unserem letzten Beispiel, sowie den Exponenten - hier die -1 - eindeutig gekennzeichnet. Diese nennt man auch normalisierte Darstellung. Dabei ist der Faktor vor dem Exponenten immer ein Wert zwischen 1 und der Basis des Zahlensystems, in unserem Falle also zehn. Für diese Zahlen gelten die bekannten Rechenregeln aus der Mathematik: z.B. können zwei normalisierte Fließkommazahlen dadurch miteinander multipliziert werden, daß man die Mantissen multipliziert und die Exponenten einfach addiert. Ergibt sich bei den Mantissen ein Wert von größer als zehn, so wird wieder der Faktor zehn in

den Exponenten übernommen. Wenn wir die letzten beiden Beispielzahlen miteinander multiplizieren, sieht das so aus:

$5 * 10^0$ mal $7 * 10^{-1}$

Durch Multiplikation der Mantissen erhält man 35, die Exponenten ergeben addiert -1. Das Ergebnis ist also $35 * 10^{-1}$. Diese Zahl muß also noch normalisiert werden. Man erhält dann $3.5 * 10^0$; also 3.5. Das Normalisieren kann man sich einfach als Verschieben des Kommas vorstellen. Wir haben also in unserer Zahl das Komma um eine Stelle nach links verschoben und zum Ausgleich dafür den Exponenten um eins erhöht. Analog muß beim Verschieben des Kommas nach rechts der Exponent um eins vermindert werden.

Wollen wir unsere Zahlen addieren, so wissen wir aus der Mathematik, daß nur Zahlen mit gleichen Exponenten addiert werden können. Die Exponenten müssen also zuerst angeglichen werden.

Wenn wir uns auf den größten Exponenten einigen, sieht das so aus:

Aus $7 * 10^{-1}$ wird $0.7 * 10^0$. Nun brauchen wir nur die Mantissen zu addieren:

$5 + 0.7 = 5.7 * 10^0$

Da die Zahl schon normalisiert ist, haben wir als Ergebnis 5.7 mal 10^0 oder einfach 5.7. Wollen wir dieses Verfahren auch auf einen Mikroprozessor übertragen, so müssen wir uns Gedanken machen, wie wir dies am besten realisieren können.

Da der Prozessor jedoch besser mit Binärzahlen umgehen kann, wollen wir einmal sehen, ob wir dieses Prinzip auch auf die Binärzahlen übertragen können.

Wir wählen also als Basis unseres Zahlensystems die 2. Bevor wir nun Fließkommazahlen auf dem Mikroprozessor implementieren, sollten wir uns zuerst einmal Gedanken machen, welchen Wertebereich unsere Zahlen überschreiten und mit welcher Genauigkeit die Zahlen gespeichert werden sollen. Beim Betrachten der Exponentialdarstellung wird uns schnell klar, daß der Exponent für den Wertebereich zuständig ist, während die Mantisse entscheidet, wieviel Stellen einer Zahl noch dargestellt werden können. Zum Problem der Genauigkeit und der Darstellbarkeit von Dezimalzahlen im Fließkommaformat werden wir später noch ausführlicher kommen.

Eine Fließkommazahl in Binärdarstellung hat also folgendes Aussehen:

$$1.011101 * 2^{10010}$$

oder $1.011101 * 2^{18}$

das sind also

$$
\begin{aligned}
& 1 * 2^{18} = 262144 \\
+ & 0 * 2^{17} = 0 \\
+ & 1 * 2^{16} = 65536 \\
+ & 1 * 2^{15} = 32768 \\
+ & 1 * 2^{14} = 16384 \\
+ & 0 * 2^{13} = 0 \\
+ & 1 * 2^{12} = 4096 \\
\hline
& = 380928
\end{aligned}
$$

Auch gebrochene Binärzahlen können so verwendet werden, z.B.

$1.011 * 2^0$

$$
\begin{aligned}
& 1 * 2^0 = 1 \\
+ & 0 * 2^{-1} = 0 \\
+ & 1 * 2^{-2} = 0.25 \\
+ & 1 * 2^{-3} = 0.125 \\
\hline
& \qquad 1.375
\end{aligned}
$$

Wollen wir aber Zahlen darstellen, die kleiner als eins sind, deren Exponent also kleiner als Null ist, so müssen wir eine Form finden, in der wir solche Exponenten ablegen können. Dazu erinnern wir uns, wie wir sonst negative Zahlen gespeichert haben. Eine Möglichkeit dazu ist das Zweierkomplement. Wenn wir für unseren Exponenten ein Byte, also 8 Bit bereitstellen, so könnten wir Zweierexponenten von -128 bis +127 darstellen. Welcher Zahlenbereich läßt sich damit darstellen? Dazu brauchen wir bloß die entsprechenden Zweierpotenzen bilden:

$2^{127} = 1.7 * 10^{38}$
$2^{-128} = 3.9 * 10^{-39}$

Wenn wir also ein Byte für den Exponenten reservieren und mit Zweierexponenten von -128 bis 127 arbeiten, lassen sich damit Zahlen darstellen, die im Dezimalsystem 38 Stellen vor dem Komma haben bzw. die erst auf der 39. Stelle hinter dem Komma beginnen. Mit diesen Zahlen überschreiten wir wohl den Bereich, der in der normalen Rechenpraxis vorkommt.
Der Commodore 64 verwendet bei seinen Fließkommazahlen nicht das Zweierkomplement zur Darstellung des Exponents, sondern einen Offset. Dazu addiert man zu jedem Exponenten die Zahl 129 oder hex $81 und betrachtet das Ergebnis als vorzeichenlose positive Zahl. In der Praxis bedeutet dies

eine Vereinfachung bei der Manipulation der Exponenten. Die folgende Tabelle gibt die Zuordnung des gespeicherten Exponenten zum echten Zweierexponenten wieder. Wir benutzen dazu der Einfachheit halber die Hexadezimaldarstellung.

Darstellung	Exponent	Wert
\$00	siehe Text	0
\$01	-128	$3.9 * 10^{-40}$
\$02	-127	$5.9 * 10^{-39}$
\$03	-126	$1.2 * 10^{-38}$
\$7F	-2	0.25
\$80	-1	0.5
\$81	0	1
\$82	1	2
\$83	2	4
\$FE	125	$4.3 * 10^{37}$
\$FF	126	$8.5 * 10^{37}$

Ist der gespeicherte Wert für den Exponenten null, so ist vereinbarungsgemäß die Zahl gleich null.

Nachdem wir den Exponenten abgehandelt haben, können wir uns um die Mantisse Gedanken machen.

Da die Mantisse über die Rechengenauigkeit entscheidet, müssen wir bestimmen, wieviel Bytes zur Speicherung der Mantisse benutzt werden sollen. Nun - der Commodore 64 verwendet dazu 4 Bytes. Wir können damit also 32 Binärziffern darstellen. Welcher Genauigkeit einer Dezimalzahl entspricht dies nun?

Dazu vergleichen wir die dezimalen Werte zweier binärer Fließkommazahlen, die sich in der letzten Stelle unterscheiden.

1.111 1111 1111 1111 1111 1111 1111 1111

und

1.111 1111 1111 1111 1111 1111 1111 1110

Die beiden Zahlen unterscheiden sich also in der letzten Stelle, die einen Wert von 2^{-31} hat. Das ist dezimal ca.

$4.6566129 * 10^{-10}$

oder

$0.46566129 * 10^{-9}$

Die beiden Zahlen haben einen Wert von etwas unter 2; sie unterscheiden sich um 5 Einheiten der zehnten Dezimalstelle. Wir können also davon ausgehen, daß wir mit einer Mantisse von 4 Byte eine dezimale Genauigkeit von etwa 9 Stellen erhalten. Dies dürfte für die meisten Anwendungen ausreichend sein. Die Genauigkeit von 9 Stellen ist eine relative Genauigkeit und unabhängig vom Exponenten. Wenn wir die dezimalen Zahlen normieren, d.h. vor dem Komma steht eine Ziffer zwischen 1 und 9, so können wir noch Zahlen, die sich in der neunten Stelle hinter dem Komma unterscheiden, mit unseren binären Fließkommazahlen unterscheiden.

Bis jetzt haben wir also die Möglichkeit, einen Exponenten zwischen -128 und +126 zu benutzen sowie eine Mantisse mit 4 Bytes, die eine dezimale Genauigkeit bis auf die neunte Stelle erlaubt. Was uns noch fehlt, ist die Möglichkeit, das Vorzeichen der Mantisse mit einzubeziehen. Durch einen kleinen Trick können wir das Vorzeichen noch in die Mantisse mit einbeziehen, ohne daß wir an Genauigkeit verlieren.

Unsere Mantisse wird immer normiert dargestellt, d.h. vor dem Komma erscheint immer eine Ziffer zwischen 1 und eins weniger als die Basis des Zahlensystems. Beim Binärsystem mit der Basis 2 kann also immer nur eine 1 erscheinen. Das machen wir uns zunutze und speichern diese eins nicht mit ab, sondern verwenden dieses Bit für das Vorzeichen. Dabei gilt die übliche Konvention, daß eine "0" eine positive Zahl bedeutet, während eine "1" eine negative Zahl kennzeichnet.

Jetzt haben wir alle Informationen, die wir benötigen, um Dezimalzahlen ins binäre Fließkommaformat umzuwandeln. Probieren wir es einmal mit verschiedenen Zahlen.

$1 = 1 * 2^0$

$= 1.000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000 * 2^0$

Wir ersetzen jetzt die erste 1 vor dem Komma gegen das Vorzeichen, berücksichtigen noch den Offset beim Exponenten und erhalten

0000 0000 0000 0000 0000 0000 0000 0000 1000 0001

Wenn wir jetzt noch den Exponenten zuerst schreiben, so wie das bei der Abspeicherung von Fließkommazahlen im Rechner geschieht, erhalten wir das folgende Bild:

1000 0001 0000 0000 0000 0000 0000 0000 0000 0000

Der besseren Übersicht wegen wandeln wir die Binär- in die Hexadezimaldarstellung um.

81 00 00 00 00

Dies ist die Darstellung der Fließkommazahl eins. Versuchen wir es jetzt mal mit der Zahl 10. Die Zerlegung in Zweierpotenzen sieht so aus:

$$
\begin{aligned}
10 &= 8 + 2 \\
&= 2^1 + 2^3 \\
&= 1 * 2^3 + 0 * 2^2 + 1 * 2^3 \\
&= 1.01 * 2^3 \text{ binär}
\end{aligned}
$$

Mit Exponent und vollständiger Mantisse erhalten wir folgendes Ergebnis:

1000 0100 0010 0000 0000 0000 0000 0000 0000 0000

bzw.

84 20 00 00 00

Nehmen wir jetzt eine negative Zahl, -5.5

$$
\begin{aligned}
-5.5 &= - (4 + 1 +0.5) \\
&= - (2^2 + 2^0 +2^{-1}) \\
&= - (1 * 2^2 + 0 * 2^1 + 1 * 2^0 + 1 * 2^{-1}) \\
&= - 1.011 * 2^2 \text{ binär}
\end{aligned}
$$

=> 1000 0011 1011 0000 0000 0000 0000 0000 0000 0000

bzw.

83 B0 00 00 00

Negative Zahlen erkennt man also daran, daß das erste Byte der Mantisse größer oder gleich $80 ist.

Mit diesen Erkenntnissen können wir leicht den dezimalen Wert jeder Fließkommazahl berechnen. Wenn wir die einzelnen Bytes wie folgt bezeichnen,

```
EX M1 M2 M3 M4
83 B0 00 00 00
```

gibt uns diese Formel den Wert:

```
X = (-SGN (M1 AND 128)*2+1)* 2↑(EX-129) *
    (1+ ((M1 AND 127)+(M2+(M3+M4/256)/256)/256)/128)
```

Sie sehen deutlich, daß das Vorzeichen aus dem obersten Bit des höchstwertigsten Bytes der Mantisse (M1) geholt wird. Beim Zweierexponenten wird der Offset von 129 berücksichtigt. Bei der Mantisse selbst wird die unterschiedliche Gewichtung der einzelnen Bytes berücksichtigt; das jeweils folgende Byte hat nur ein 256tel des Wertes des vorangegangenen Bytes. Probieren wir unsere Formel einmal mit der letzten Fließkommazahl aus.

```
X = (-SGN (176 AND 128)*2+1)* 2↑(131-129) *
    (1+ ((176 AND 127)+(0+(0+0/256)/256)/256)/128)
```

Sie sehen, wir erhalten wieder unseren Wert von -5.5.

Bis jetzt haben wir bei der Umwandlung von Dezimalzahlen in binäre Fließkommazahlen noch keine Probleme gehabt. Versuchen wir jetzt mal, den Wert 0.4 umzuwandeln.

Wir gehen dabei systematisch vor und ziehen jeweils die größte Zweierpotenz ab, die in der Zahl enthalten ist.

```
  0.4               Zweierpotenz
- 0.25                  -2
 =====
  0.15
- 0.125                 -3
 ======
  0.025
- 0.015625              -6
 =========
  0.009375
- 0.0078125             -7
 ==========
  0.0015625
- 0.0009765625          -10
 ============
  0.0005859375
- 0.00048828125         -11
 ==============
  0.00009765625
- 0.00006103515625      -14
 =================
  0.00003662109275      usw.
```

Diese Rechnung können wir beliebig fortführen, die Zahlenfolge bei der Konvertierung bricht nicht ab. Wir erhalten den periodischen Wert

1. 1001 1001 1001 1001 1001 1001 1001 1001 1001 ... * 2^{-2}

Wir können die Zahl 0.4 also nicht exakt als binäre Fließkommazahl darstellen. Wir müssen unsere Ziffernfolge bei der 31. Stelle hinter dem Komma abbrechen und erhalten dann

1. 1001 1001 1001 1001 1001 1001 1001 100 * 2^{-2}

Um die Genauigkeit etwas zu erhöhen, schneiden wir die Ziffernfolge nicht einfach ab, sondern runden auf oder ab. Bei binären Werten wird dann aufgerundet, wenn die folgende Ziffer eine eins ist; bei einer null bleibt die Zahl bestehen. In unserem Falle müssen wir also aufrunden.

1. 1001 1001 1001 1001 1001 1001 1001 101 * 2^{-2}

Wenn wir jetzt noch den Exponenten und das Vorzeichen berücksichtigen, erhalten wir die Folge

0111 1111 0100 1100 1100 1100 1100 1100 1100 1101

oder hexadezimal

7F 4C CC CC CD

Daß man mit binären Fließkommazahlen nicht alle Dezimalzahlen exakt darstellen kann, liegt nicht an der Basis 2, sondern ist ein generelles Phänomen bei der Umwandlung von einem Zahlensystem in ein anderes. Versuchen Sie einmal, den Bruch 1/3 im Dezimalsystem darzustellen - es wird Ihnen nicht exakt gelingen. Die Ziffernfolge

0.33333 33333 33333 .....

muß irgendwann abgebrochen werden. In einem Zahlensystem mit der Basis 3 ist dies jedoch ohne weiteres möglich - wir erhalten einfach

0.1

was wir als 1 * 3^{-1} gleich ein Drittel interpretieren.

Nachdem wir nun etwas über die Grundlagen der Fließkommazahlen gehört haben, wollen wir sie jetzt praktisch nutzen. Da ein Großteil des eingebauten BASIC-Interpreters sich mit der Konvertierung der verschiedenen Zahlenformate sowie mit der Fließkommaarithmetik beschäftigt, liegt es nahe, sich diese Routinen zunutze zu machen.

Der BASIC-Interpreter hat zwei sogenannte Fließkommaakkumulatoren (floating point accu), meist kurz FAC genannt, in denen die Zahlen zur Verarbeitung gespeichert werden. Der FAC#1 wird bei jeder Operation benutzt; benötigt eine Operation, wie z.B. die Addition, zwei Operanden, so steht der zweite im FAC#2. Das Ergebnis wird immer im FAC#1 abgelegt. Oft wird der Fließkommaakku#1 auch nur als FAC bezeichnet und FAC#2 wird ARG (Argument) genannt. In diesen Fließkommaakkumulatoren werden die Zahlen nicht in der verkürzten 5-Byte-Form abgespeichert, sondern für das Vorzeichen wird ein zusätzliches Byte verwendet. Die sonst durch das Vorzeichen ersetzte eins vor dem Komma wird dabei wieder rekonstruiert. Zusätzlich wird noch ein Rundungsbyte verwendet, um bei verschiedenen Operationen eine Rundung zu ermöglichen. Die Fließkommaakkumulatoren benutzen die folgenden Speicherstellen in der Zeropage:

	FAC	ARG
Exponent	$61	$69
Mantisse 1	$62	$6A
Mantisse 2	$63	$6B
Mantisse 3	$64	$6C
Mantisse 4	$65	$6D
Vorzeichen	$66	$6E
Rundungsbyte	$70	
Vorzeichen-vergleichsbyte	$6F	

Das Vorzeichenvergleichsbyte wird bei Operationen mit zwei Operanden benötigt und ist $00 bei gleichen Vorzeichen und $FF bei ungleichen Vorzeichen.

Der BASIC-Interpreter hat eine Vielzahl von Routinen, die mit den Fließkommazahlen hantieren. Fangen wir zuerst mit der Routine an, die eine Dezimalzahl liest und in eine Fließkommazahl umwandelt. Diese Routine wird bei jeder Zahleneingabe benutzt. Vorher sehen wir uns noch kurz eine Routine namens 'CHRGET' an, die ein beliebiges Zeichen aus einer eingegebenen Zeile oder aus dem BASIC-Text liest. Die Routine steht ebenfalls in der Zeropage und hat die Aufgabe, ein Zeichen zu lesen und verschiedene Prüfungen durchzuführen. Die Routine hat noch einen zweiten Einsprungpunkt mit dem Namen 'CHRGOT', über den das zuletzt gelesene Zeichen nochmals geholt werden kann.

```
CHRGET  INC TXTPTR
        BNE CHRGOT
        INC TXTPTR+1
CHRGOT  LDA TEXT
        CMP #":"
        BCS EXIT
        CMP #" "
        BEQ CHRGET
        SEC
        SBC #$30
        SEC
        SBC #$D0
EXIT    RTS
```

Der Trick bei dieser Routine, der auch der Grund dafür ist, daß sie im RAM stehen muß, besteht in der Selbstmodifiktion. Die Adresse TXTPTR, der Zeiger auf die aktuelle Position, von

der das Zeichen geholt werden soll, befindet sich in der Routine selbst. Das wird sofort deutlich, wenn wir uns den Opcode ansehen.

```
0073  E6 7A       INC $7A
0075  D0 02       BNE $0079
0077  E6 7B       INC $7B
0079  AD 02 02    LDA $0202
007C  C9 3A       CMP #$3A
007E  B0 10       BCS $008A
0080  C9 20       CMP #$20
0082  F0 EF       BEQ $0073
0084  38          SEC
0085  E9 30       SBC #$30
0087  38          SEC
0088  E9 D0       SBC #$D0
008A  60          RTS
```

Wenn wir die Routine CHRGET aufrufen, wird also zuerst das Adressfeld des Ladebefehls an der Adresse CHRGOT um eins erhöht und dann der Inhalt dieser Speicherstelle gelesen. Nun folgen verschiedene Prüfungen. Zuerst wird mit dem Doppelpunkt verglichen. Ist der ASCII-Kode des gelesenen Zeichens größer oder gleich, so wird direkt zur RTS-Anweisung verzweigt. Es ist also das Carry-Flag gesetzt. War das Zeichen ein Doppelpunkt, so ist zusätzlich noch das Zero-Flag gesetzt. Da der Doppelpunkt das Ende einer Anweisung kennzeichnet, kann dies leicht mit Hilfe des Zero-Flags getestet werden. Ist das Zeichen kleiner als der Doppelpunkt, so wird als nächstes mit dem Leerzeichen verglichen. Fällt der Vergleich positiv aus, so wird wieder nach CHRGET verzweigt, also das nächste Zeichen geholt. Leerzeichen werden also grundsätzlich vom Interpreter überlesen. Die nächsten beiden Subtraktionen verändern den Wert nicht,

sondern haben lediglich die Aufgabe, das Carry-Flag zu beeinflußen. Das Carry-Flag wird immer dann gelöscht, wenn das gelesene Zeichen eine ASCII-Ziffer zwischen "0" und "9" entsprechend $30 und $39 war.

Fassen wir die Ergebnisse noch einmal zusammen: Die CHRGET-Routine erhöht den Textzeiger TXTPTR und übergibt das Zeichen im Akku. Handelt es sich dabei um einen Doppelpunkt oder ein Nullbyte, die das Ende eines Statements oder einer Zeile anzeigen, so ist das Zero-Flag gesetzt; war das gelesene Zeichen eine Ziffer, so ist das Carry-Flag gelöscht.

Doch kommen wir jetzt wieder zu unserer Konvertierungsroutine. Ehe wir diese Routine aufrufen können, muß der Akku das erste Zeichen der Zahl enthalten und die Flags müssen entsprechend der CHRGET-Routine gesetzt sein. Der Textzeiger TXTPTR muß natürlich auf unsere Zahl zeigen. Das folgende kleine Programm liest eine Zahl ein und konvertiert diese ins Fließkommaformat.

```
033C                        .OPT P,00
033C                        *=   828
007A            TXTPTR      =    $7A
0079            CHRGOT      =    $79
BCF3            ASCFLOAT =       $BCF3
033C A9 4B                  LDA  #<ZAHL
033E A0 03                  LDY  #>ZAHL
0340 85 7A                  STA  TXTPTR
0342 84 7B                  STY  TXTPTR+1
0344 20 79 00               JSR  CHRGOT
0347 20 F3 BC               JSR  ASCFLOAT
034A 00                     BRK
034B 31 2E 32 ZAHL          .ASC "1.2345"
0351 00                     .BYT 0
```

Wenn wir diese Routine assemblieren und vom Monitor aus mit

```
G 033C
```

starten, so wird die Zahl 1.2345 ins Fließkommaformat gewandelt und im FAC#1 abelegt, den wir uns mit

```
M 0061 0066
```

ansehen können. Wir erhalten die folgenden Werte:

```
>: 0061 81 9E 04 18 93 00
```

Probieren wir nochmal unsere 0.4. Dazu müssen wir ab Adresse $034B die Ziffernfolge ablegen und durch ein Nullbyte abschließen:

```
M 034B 034B
>: 034B 30 2E 34 00
```

Wir erhalten als Ergebnis

```
>: 0061 7F CC CC CC CC 00
```

Das Vorzeichen ist als sechstes Byte seperat abgespeichert und ist bei positiven Zahlen null. Bei dieser Zahlenkonvertierung können wir auch Zahlen mit Zehnerexponenten verabeiten, z.B. -1.4E-7 oder 1E12. Nehmen wir als nächstes Beispiel eine negative Zahl, z.B. -1E8. Jetzt erhalten wir

```
>: 0061 9B BE BC 20 00 FF
```

Diesmal wird das negative Vorzeichen durch $FF gekennzeichnet.

Betrachten wir noch einmal kurz das Ergebnis beim Wert von 0.4. Wir haben wir einen Wert erhalten, der um eine Einheit der letzten Stelle kleiner ist als dies bei der manuellen Umwandlung der Fall war. In unserer Routine wird keine automatische Rundung vorgenommen, es wird lediglich im Rundungsbyte vermerkt, ob ein Übertrag in die nächsten Stellen vorhanden ist. Setzen Sie nochmal 0.4 ein und schauen in Adresse $70 nach, welchen Wert das Rundungsbyte hat. Wir erhalten $80. Das bedeutet, daß das Ergebnis um eine Einheit der letzten Stelle erhöht werden muß. Auch dafür steht bereits eine Routine zur Verfügung. Hängen wir diese noch an unser kleines Programm an, so wird der konvertierte Wert automatisch gerundet.

```
033C                      .OPT P,00
033C                      *=   828
007A          TXTPTR      =    $7A
0079          CHRGOT      =    $79
BCF3          ASCFLOAT    =    $BCF3
BC1B          ROUND       =    $BC1B
033C A9 4E                LDA  #<ZAHL
033E A0 03                LDY  #>ZAHL
0340 85 7A                STA  TXTPTR
0342 84 7B                STY  TXTPTR+1
0344 20 79 00             JSR  CHRGOT
0347 20 F3 BC             JSR  ASCFLOAT
034A 20 1B BC             JSR  ROUND
034D 00                   BRK
034E 31 2E 32 ZAHL        .ASC "0.4"
0351 00                   .BYT 0
```

Schauen wir uns den FAC an, so haben wir das gewünschte Ergebnis.

```
>: 0061 7F CC CC CC CD 00
```

Durch das Runden wird natürlich das Rundungsbyte gelöscht, wovon Sie sich leicht überzeugen können.

Nachdem wir Ziffernstrings in das interne Fließkommaformat wandeln können, lernen wir jetzt das umgekehrte Verfahren kennen, die Umwandlung vom Fließkommaformat zurück in einen String mit Dezimalziffern. Auch dafür sind bereits Routinen vorhanden. Diese Aufgabe übernimmt die Routine FLOATASC mit der Adresse $BDDD. Durch den Aufruf dieser Routine wird die Umwandlung in einen Zahlenstring veranlaßt, der ab der Adresse $0100 abgelegt wird. Probieren wir das einmal aus. Dazu schreiben wir folgende Werte in den FAC:

```
>: 0061 90 8F 00 00 00 80
```

Schauen wir uns nach dem Aufruf der Routine das Ergebnis an,

```
>M 0100 0107
>: 0100 20 33 36 36 30 38 00    -36608
```

Der obige Wert im FAC bedeutet also die Dezimalzahl -36608. Nach dem Aufruf dieser Routine enthalten Akku und Y-Register immer die Adresse, wo der String abgelegt wurde, hier konstant A=0 und Y=1 (low/high-Byte). Jetzt können wir den String z.B. auf dem Bildschirm ausgeben. Auch dazu können wir eine vorhandene BASIC-Routine benutzen: STROUT mit der Adresse $AB1E.

Bevor wir dazu kommen, mit unseren Fließkommazahlen zu rechnen, wollen wir erst noch die verschiedenen Routinen des BASIC-Interpreters kennenlernen, die eine Konversion von verschiedenen Ganzzahlformaten ins Fließkommaformat ermög-

lichen. Dies ist besonders für unsere eigenen Maschinenprogramme wichtig, da zum einen sämtliche Arithmetik des BASIC-Interpreters mit den Fließkommazahlen vonstatten geht, Ein- und Ausgaben für diese Routinen aber oft im INTEGER-Format bereitgestellt bzw. erwartet werden.

1.2. Umwandlung ins Fließkommaformat
Ein-Byte-Wert mit Vorzeichen

Mit der folgenden Routine kann ein vorzeichenbehafteter Ein-Byte-Wert ins Fließkommaformat gewandelt werden. Es kann also ein Ergebnis zwischen -128 und +127 auftreten. Der Byte-Wert wird im Akku übergeben.

```
LDA #BYTE
JSR $BC3C
```

Ein Wert von $80 wird zu -128, $FF wird zu -1 und $7F wird zu 127 konvertiert.

Ein-Byte-Wert ohne Vorzeichen

Soll das Vorzeichen nicht berücksichtigt werden, so muß zur Konvertierung folgende Routine benutzt werden:

```
LDY #BYTE
JSR $B3A2
```

Mit dieser Routine wird aus $00 null, $80 wird zu 128 und aus $FF wird 255.

Zwei-Byte-Wert mit Vorzeichen

Ein Zwei-Byte-Wert mit Vorzeichen läßt sich mit der folgenden Routine ins Fließkommaformat wandeln:

```
LDY #LOW
LDA #HIGH
JSR $B395
```

Das niederwertige Byte muß also im Y-Register bereitgestellt werden, während im Akku das höherwertige Byte steht.

Folgende Beispiele demonstrieren die Umwandlung:

A	Y	Fließkommawert
00	00	0
00	01	1
00	FF	255
01	00	256
7F	FF	32767
80	00	-32768
FF	FF	-1

Zwei-Byte-Wert ohne Vorzeichen

Soll das Vorzeichen nicht beachtet werden, kommt folgende Routine zum Einsatz:

```
LDY #LOW
LDA #HIGH
STY $63
STA $62
LDX #$90
SEC
JSR $BC49
```

Bei dieser Umwandlung wird das Vorzeichen nicht berücksichtigt und wir bekommen Werte von 0 bis 65535.

A	Y	Fließkommawert
00	00	0
00	01	1
00	FF	255
01	00	256
7F	FF	32767
80	00	32768
FF	FF	65535

Drei-Byte-Werte mit Vorzeichen

Obwohl in der Praxis kaum mit 3-Drei-Werten gearbeitet wird, sollen trotzdem die Routinen zur Umwandlung solcher Daten ins Fließkommaformat erwähnt werden.

```
LDA #LOW
LDX #MID
LDY #HIGH
STY $62
STX $63
STA $64
LDA $62
EOR #$FF
ASL  A
LDA #0
STA $65
LDX #$98
JSR $BC4F
```

Die Konversionstabelle sieht so aus:

Y X A	Fließkommawert
00 00 00	0
00 00 FF	255
00 FF FF	65535
7F FF FF	8388607
80 00 00	-8388608
FF FF FF	-1

Wir können mit 3-Byte-Werten bzw. 24-Bit-Zahlen den Wertebereich von -8 388 608 bis 8 388 607 abdecken.

Drei-Byte-Werte ohne Vorzeichen

Soll das Vorzeichen nicht verwendet werden, so kann folgende Routine benutzt werden.

```
LDA #LOW
LDX #MID
LDY #HIGH
JSR $AF87
JSR $AF7E
```

Hier können wir Werte zwischen 0 und $2^{24}-1$ = 16 777 215 darstellen.

Y X A	Fließkommawert
00 00 00	0
00 00 FF	255
00 FF FF	65535
7F FF FF	8388607
80 00 00	8388608
FF FF FF	16277215

Vier-Byte-Werte mit Vorzeichen

Der Vollständigkeit halber soll auch noch die Konversion von 32-Bit INTEGER-Werten durchgeführt werden. Hier sehen die Routinen ähnlich aus. Da hier vier Bytes übergeben werden müssen, geht die Routine davon aus, daß diese Werte schon im FAC von Adresse $62 (höchstwertiges Byte) bis $65 (niederwertigstes Byte) stehen.

```
LDA $62
EOR #$FF
ASL  A
LDA #0
LDX #$A0
JSR $BC4F
```

Wir erhalten dann folgende Umwandlungstabelle.

```
$62 63 64 65   Fließkommawert
 00 00 00 00   0
 00 00 00 FF   255
 00 00 FF FF   65535
 00 FF FF FF   16777215
 7F FF FF FF   2147483647    (2.14748365E+09)
 80 00 00 00   -2147483648  (-2.41748365E+09)
 FF FF FF FF   -1
```

Vier-Byte-Werte ohne Vorzeichen

Zum Schluß soll auch diese Konversionsroutine vorgestellt werden. Auch hier müssen die Werte schon im FAC vorliegen.

```
SEC
LDA #0
```

```
LDX #$A0
JSR $BC4F
```

Hier können wir den Wertebereich von 0 bis $2^{32}-1$ = 4 294 967 295 verwenden.

```
$62 63 64 65   Fließkommawert
 00 00 00 00   0
 00 00 00 FF   255
 00 00 FF FF   65535
 00 FF FF FF   16777215
 7F FF FF FF   2147483647     (2.14748365E+09)
 80 00 00 00   2147483648     (2.41748365E+09)
 FF FF FF FF   4294967295     (4.2949673E+09)
```

Die bis jetzt besprochenen Routinen sind nützlich, wenn man Ein- bis Vier-Byte-Werte aus eigenen Maschinenroutinen als Argumente für die Fließkommaroutinen des BASIC-Interpreters benutzen will. Der umgekehrte Wert - die Konvertierung von Fließkommawerten in INTEGER-Zahlen - soll nun besprochen werden.

1.3 Umwandlung ins INTEGER-Format

Für die Umwandlung vom Fließkomma- ins INTEGER-Format benötigt man nur eine Routine. Das Ergebnis dieser Umwandlung ist grundsätzlich eine 4-Byte-Zahl mit Vorzeichen. Wenn die umzuwandelnde Zahl im FAC steht, genügt der Aufruf von

```
JSR $BC9B
```

um die Konvertierung durchzuführen. Da sich nur Zahlen kleiner als 2^{31} fehlerfrei in INTEGER-Werte wandeln lassen, sollte vorher der Exponent der Zahl darauf überprüft werden, ob er kleiner als $A0 ist. Das Ergebnis der Umwandlung steht dann in $62 (höchstwertiges Byte inklusive Vorzeichen) bis $65 (niederwertigstes Byte). Sehen wir uns ein Beispiel an.

Der FAC soll den Fließkommawert 10 enthalten:

```
          EX M1 M2 M3 M4 SGN
>: 0061   84 A0 00 00 00 00
```

Nach dem JSR $BC9B erhalten wir

```
>: 0061   84 00 00 00 0A 00
```

Enthält der FAC keinen ganzzahligen Wert, so wird der Nachkommaanteil wie bei der INTEGER-Funktion abgeschnitten, z.B. wenn im FAC 321.25 steht:

```
          EX M1 M2 M3 M4 SGN
>: 0061   89 A0 A0 00 00 00
```

Als Ergebnis erhalten wir

```
>: 0061 89 00 00 01 41 00
```

also $41 + $100 = 65 + 256 = 321. Bei negativen gebrochenen Zahlen wird ebenfalls zum nächstkleineren ganzzahligen Wert hin abgeschnitten, so wird aus -0.5 eine -1.

```
         EX M1 M2 M3 M4 SGN
>: 0061 80 80 00 00 00 FF
```

Als Ergebnis erhalten wir

```
>: 0061 80 FF FF FF FF FF
```

also eine minus eins.

Bei den Routinen des BASIC-Interpreters werden wir später noch Routinen kennenlernen, die vor der Umwandlung ins INTEGER-Format noch Bereichsprüfungen vornehmen, z.B. auf den Bereich 0 bis 255 oder -32768 bis 32767.

1.4 Die Rechenroutinen des BASIC-Interpreters

Nachdem wir uns bis jetzt mit der Zahlenein- und ausgabe sowie mit der Konvertierung von Zahlen beschäftigt haben, wird es Zeit, daß wir die ersten Berechnungen durchführen.

Der Interpreter kennt fünf arithmetische Grundoperationen mit zwei Operanden, das sind Addition, Subtraktion, Multiplikation, Division und Potenzierung. Wenn wir diese Funktionen benutzen wollen, so muß der erste Operand im FAC stehen, während der zweite in ARG erwartet wird. Nach dem Aufruf der Routine steht das Ergebnis im FAC. Hier die Adressen der Routinen:

```
ADDITION            FAC := ARG + FAC      $B86A
SUBTRAKTION         FAC := ARG - FAC      $B853
MULTIPLIKATION      FAC := ARG * FAC      $BA2B
DIVISION            FAC := ARG / FAC      $BB12
POTENZIERUNG        FAC := ARG ^ FAC      $BF7B
```

Vor dem Aufruf dieser Routinen muß der Akku den Exponenten von FAC ($61) enthalten. Ist dieser Exponent null, so ist vereinbarungsgemäß auch der Wert im FAC gleich null und es können Sonderfälle abgehandelt werden (ARG + 0 = ARG; ARG * 0 = 0; ARG / 0 erzeugt 'DIVISION BY ZERO'; ARG ^ 0 ergibt 1). Wenn wir später Routinen benutzen, die FAC und ARG mit Werten versorgen, so stellen diese automatisch den Exponenten im Akku zur Verfügung. Doch versuchen wir einmal zwei Werte zu multiplizieren, z.B. 7*13 = 91.

```
 7 =  83 E0 00 00 00 00
13 =  84 D0 00 00 00 00
```

Wir versorgen die Fließkommaakkumulatoren mit den Werten, laden den Akku mit dem Exponenten von FAC und rufen die Routine auf.

```
>: 0061 83 E0 00 00 00 00
>: 0069 84 D0 00 00 00 00

>, 1000    A5 61       LDA $61
>, 1002    20 2B BA    JSR $BA2B
>, 1005    00          BRK

>G 1000

B*
    PC  IRQ  SR AC XR YR SP   NV-BDIZC
```

```
>; 1006 EA31 A0 87 B6 00 F8    10100000

>: 0061 87 B6 00 00 00 00 00
```

Wandeln wir das Ergebnis in eine Dezimalzahl um.

$1.0110110 * 2^6 = 1011011$
$= 64 + 16 + 8 + 2 + 1 = 91$

Versuchen wir nun eine Potenzierung, z.B. $3^7 = 2187$

```
3 =  82 C0 00 00 00 00
7 =  8E E0 00 00 00 00
```

Nun können wir die Werte übergeben und die Potenzierungsroutine aufrufen.

```
>: 0061 83 E0 00 00 00 00
>: 0069 82 C0 00 00 00 00

>, 1000    A5 61       LDA $61
>, 1002    20 7B BF    JSR $BF7B
>, 1005    00          BRK

>G 1000

B*
   PC  IRQ  SR AC XR YR SP   NV-BDIZC
>; 1006 EA31 22 00 61 00 F8   10100000

>: 0061 8C 88 B0 00 02 00 00
```

Wir erhalten also

1.000 1000 1011 0000 0000 0000 0000 0010 * 2^{11} =
1000 1000 1011. 0000 0000 0000 0000 0010

$= 2^{11} + 2^7 + 2^3 + 2^1 + 2^0 + 2^{-19}$
$= 2048 + 128 + 8 + 2 + 1 + 1.9*10^{-6}$
$= 2187.0000019$

Sie sehen also, daß das Ergebnis nicht exakt stimmt - es besteht eine Abweichung in der vorletzten Stelle. Da bei der Umwandlung von Binär- in Dezimalzahlen jedoch nur 9 Stellen angezeigt werden, erhalten wir bei der Anweisung

```
PRINT 3↑7
```

als Ergebnis 2187, die Rechnung

```
PRINT 3↑7 - 2187
```

hat jedoch

```
1.90734863E-06
```

zum Ergebnis und offenbart den Unterschied. Wenn wir die Routine für die Exponentiation genauer analysieren, so können wir erkennen, daß folgender Algorithmus verwendet wird:

```
A ↑ B => EXP( B * LOG( A ))
```

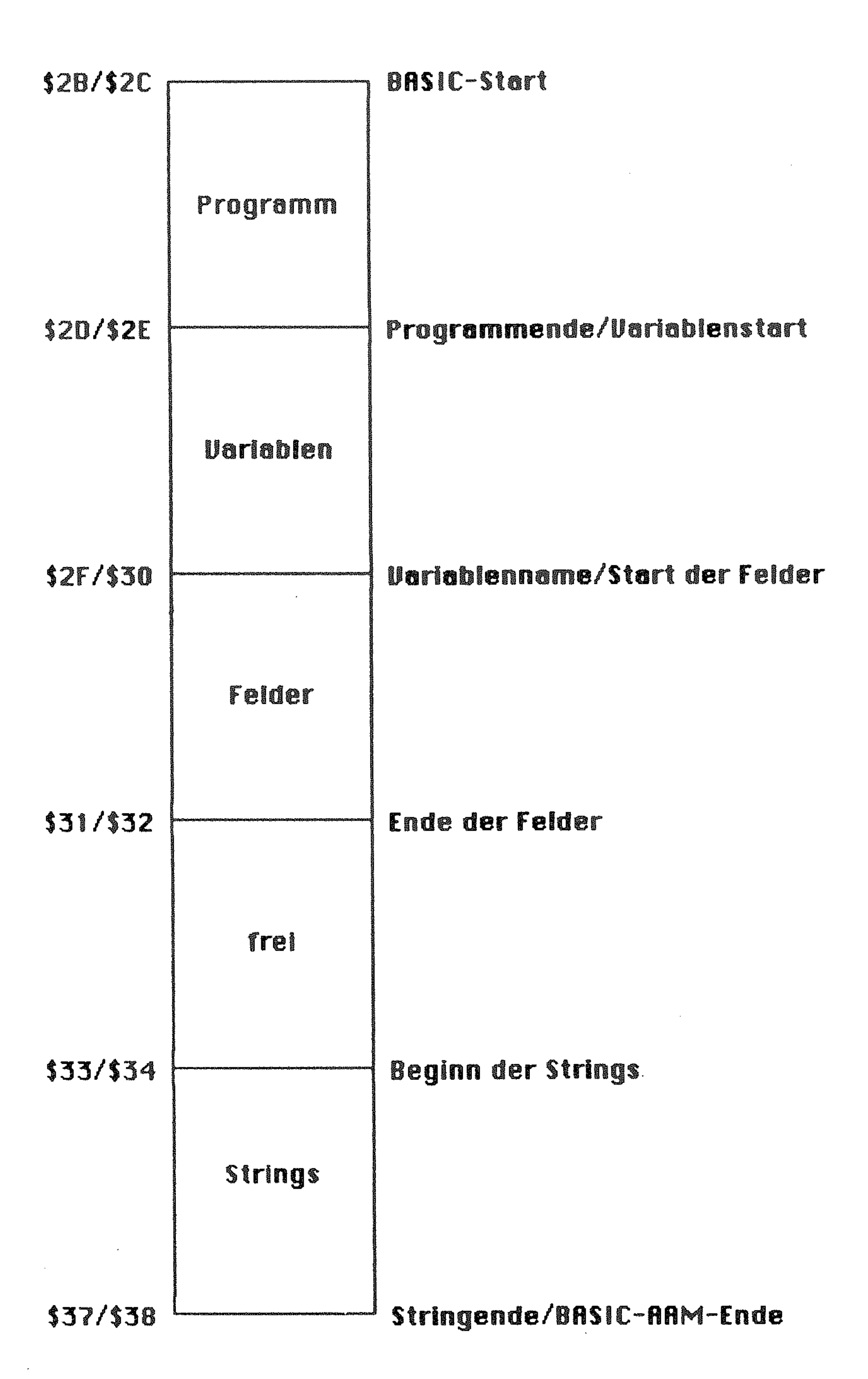
Da der BASIC-Interpreter sowohl die EXP- als auch die LOG-Funktion nur näherungsweise berechnen kann - wie wir später noch sehen werden - ist es nicht verwunderlich, daß sich dabei Abweichungen ergeben. Da für die Potenzfunktion zwei andere Funktionen berechnet werden müssen, ist diese Routine auch eine der langsamsten arithmetischen Routinen,

sie braucht im Mittel mehr als 50 Millisekunden. Deshalb empfiehlt es sich bei ganzzahligen und einfachen Exponenten, die Potenzierung auf eine Multiplikation zurückzuführen - einmal der Geschwindigkeit wegen, zum anderen der Genauigkeit halber.

3 ↑ 2 berechnet man besser als 3 * 3

Die Multiplikation ist hier mehr als 20mal schneller. Eine Übersicht über die Ausführungszeiten finden Sie später.

Damit wir unsere Kenntisse praktisch anwenden können, hier zuerst noch etwas über die Variablenverwaltung des BASIC-Interpreters. Dazu existieren in der Zeropage eine Reihe von Zeigern, die die Bereiche für BASIC-Programm, normale Variablen, indizierte Variablen und Strings bestimmen. Die entsprechenden Werte für den 128er-Modus können Sie aus der Tabelle im Anhang entnehmen. Die Aufteilung sieht folgendermaßen aus.



$2B/$2C
BASIC-Start
Programm
$2D/$2E
Programmende/Variablenstart
Variablen
$2F/$30
Variablenname/Start der Felder
Felder
$31/$32
Ende der Felder
frei
$33/$34
Beginn der Strings
Strings
$37/$38
Stringende/BASIC-RAM-Ende

Nach dem Einschalten des Rechners liegt der BASIC-Start bei \$0801 = 2049 und das BASIC-Ende auf \$A000 = 40960. Wenn Sie eine Programmzeile

```
10 A=1
```

eingeben, so wird sie folgendermaßen abgelegt:

Ab dem BASIC-Start \$0801 steht

Adresse der nächsten Zeile
Zeilennummer
Programmzeile
0

Vom Monitor aus sieht das so aus:

```
>M 0800 080F
>: 0800 00 09 08 0A 00 41 B2 31
>: 0808 00 00 00
```

Die Programmzeiger haben folgende Werte:

```
>M 002B 0037
>: 002B 01 08 0B 08 0B 08 0B 08
>: 0033 00 A0 00 00 00 A0
```

Versuchen wir einmal, diesen Inhalt zu interpretieren. Ab Adresse (\$2B/\$2C) = \$0801 steht zuerst die Adresse der nächsten Programmzeile im Format lo/hi, also \$0809. Dann folgt die Zeilennummer, ebenfalls im lo/hi-Format = \$000A = 10. Nun folgt der Programmtext \$41 = "A", \$B2 ist der Interpreterkode für "=", während \$31 die "1" im ASCII-Kode ist. Zur Kennzeichnung des Zeilenendes dient ein Nullbyte.

Die nächste Programmzeile folgt nach dem gleichen Schema. Da wir aber nur eine Programmzeile eingegeben haben, finden wir hier als Adresse der nächsten Programmzeile $0000. Damit wird vereinbarungsgemäß das Ende des Programms gekennzeichnet. Die darauffolgende Adresse $080B steht in ($2D/$2E) und kennzeichnet das Programmende und gleichzeitig den Beginn der normalen Variablen. Da wir noch keine Variablen definiert haben, haben die Zeiger für Variablenende und Feldende die gleichen Werte. Wenn wir mit RUN unser Programm laufen lassen, wird die Variable A angelegt.

```
>M 0800 0810
>: 0800 00 09 08 0A 00 41 B2 31
>: 0808 00 00 00 41 00 81 00 00
>: 0810 00 00

>M 002B 0037
>: 002B 01 08 0B 08 12 08 12 08
>: 0033 00 A0 00 00 00 A0
```

Jetzt zeigt der Variablenstart ($2D/$2E) auf $080B und das Variablenende ($2F/$30) auf $0812. Die Variablentabelle ist also $0812 - $080B = $0007 = 7 Bytes lang und hat folgenden Inhalt:

```
>: 080B 41 00 81 00 00 00 00
```

Die Einträge jeder Variablen sind generell 7 Bytes lang. Dabei bedeuten die ersten beiden Bytes den Namen der Variablen, bei uns $41 $00 = A. Variablennamen, die nur ein Zeichen lang sind, enthalten als zweites Zeichen also eine Null. Anschließend folgt die Fließkommadarstellung des Wertes, und zwar in der verkürzten 5-Byte-Form, bei der das Vorzeichen in das oberste Bit der Mantisse gepackt ist. Der Wert 81 00 00 00 00 steht also für die eins.

Sehen wir gleich nochmal, was passiert, wenn wir INTEGER-Variablen verwenden. Dazu ändern wir unsere Programmzeile in

```
10 A%=1
```

```
>M 002B 0037
>: 002B 01 08 0C 08 13 08 13 08
>: 0033 00 A0 00 00 00 A0
```

```
>M 0800 0810
>: 0800 00 0A 08 0A 00 41 25 B2
>: 0808 31 00 00 00 C1 80 00 01
>: 0810 00 00 00
```

Das Programm ist also durch das Prozentzeichen um ein Byte länger geworden. Aber sehen wir uns den Variableneintrag an. Er ist ebenfalls 7 Bytes lang. Erkennen Sie noch den Namen A bzw. A% in der Tabelle ? Nun, wenn Sie das Bitmuster $C1 $80 mit $41 $00 vergleichen, so sehen Sie, daß in beiden Bytes das oberste (siebte) Bit gesetzt ist. Damit wird eine INTEGER-Variable gekennzeichnet. Die nächsten beiden Bytes enthalten den 16-Bit INTEGER-Wert $0001, wobei zuerst das höchstwertige Byte kommt. Die nächsten drei Bytes sind bei INTEGER-Variablen ungenutzt. Wenn Sie mit normalen INTEGER-Variablen arbeiten, haben Sie dadurch also keine Speicherplatzersparnis. Auch bringen INTEGER-Variablen keine Erhöhung der Rechengeschwindigkeit, im Gegenteil - sämtliche Rechenoperationen laufen in Fließkommaarithmetik ab, so daß zusätzlich noch Konversionen erforderlich werden.

Gehen wir in diesem Zusammenhang noch kurz auf die Stringvariablen ein. Schreiben Sie dazu folgende Programmzeile:

10 A$="STRING"

Starten Sie das Programm und sehen Sie sich das Ergebnis mit dem Monitor an.

```
>M 002B 0037
>: 002B 01 08 13 08 1A 08 1A 08
>: 0033 00 A0 00 00 00 A0

>M 0800 0810
>: 0800 00 11 08 0A 00 41 24 B2
>: 0808 22 53 54 52 49 4E 47 22
>: 0810 00 00 00 41 80 06 09 08
>: 0818 00 00
```

Schauen Sie auf die Zeiger für den Stringbereich, so hat sich dort nichts getan. Die Variablentabelle beginnt bei $0813 und sieht so aus:

```
>: 0813 41 80 06 09 08 00 00
```

Die ersten beiden Bytes repräsentieren wieder den Namen der Variablen. Sicher haben Sie schon erkannt, daß zur Kennzeichnung einer Stringvariablen im zweiten Zeichen des Namens das oberste Bit gesetzt wird - aus $41 $00 wird $41 $80. Die nächsten drei Werte sind folgendermaßen zu interpretieren: Der erste Wert, $06, gibt die Länge des Strings an: 6 Zeichen. Die nächsten beiden Byte zeigen auf die Adresse, an der der String zu finden ist: $0809. Sie zeigen also ins Programm direkt hinter das erste Anführungszeichen. Das ist auch der Grund dafür, daß der Stringbereich noch leer bleibt. Anders wird es jedoch, wenn der String verändert wird, z.B.

10 A$="STRING"
20 A$=LEFT$(A$,3)

```
>M 002B 0037
>: 002B 01 08 22 08 29 08 29 08
>: 0033 FD 9F 00 A0 00 A0

>M 0800 0810
>: 0800 00 11 08 0A 00 41 24 B2
>: 0808 22 53 54 52 49 4E 47 22
>: 0810 00 20 08 14 00 41 24 B2
>: 0818 C8 28 41 24 2C 33 29 00
>: 0820 00 00 41 80 03 FD 9F 00
>: 0828 00
```

Hier beginnt die Variablentabelle bei $0822.

```
>: 0822 41 80 03 FD 9F 00 00
```

Nach dem Namen folgt wieder die Länge, diesmal 3 und anschließend die Adresse des Strings $9FFD, was gleichzeitig die Untergrenze der Strings ist. Schauen wir dort nach, so finden wir da unseren neuen String "STR".

```
>: 9FFD 53 54 52
```

Wie sind nun Variablenfelder organisiert? Löschen wir dazu unser Programm und schreiben

10 DIM A(500)

Wir erhalten folgende Speicherbelegung:

```
>M 002B 0037
```

```
>: 0028 01 08 10 08 10 08 E0 11
>: 0033 00 A0 00 00 00 A0
```

Da hierbei noch keine normalen Variablen definiert sind, haben Start- und Endvektoren den gleichen Wert von $0810. Dies ist auch der Beginn des Arraybereichs. Der Arraybereich geht bis $11E0, ist also $11E0 - $0810 = $09D0 = 2512 Bytes lang. Der Anfang sieht so aus:

```
>M 0810 0820
>: 0810 41 00 D0 09 01 01 F5 00
>: 0818 00 00 00 00 00 00 00 00
>: 0820 00 00 00 00 00 00 00 00
```

Wir erkennen in den ersten beiden Bytes wieder den Namen des Arrays A wieder. Die folgenden beiden Bytes geben den vom Array belegten Speicherplatz an, jene $09D0 Bytes, die wir oben berechnet hatten. Die nächste eins bedeutet, daß wir es mit einem eindimensionalen Array zu tun haben. Nun folgt die Anzahl der Arrayelemente $01F5 = 501. Fünfhundertundeins deshalb, weil auch ein Element mit dem Index 0 existiert A(0) - A(500). Anschließend folgen die Werte des Feldes beginnend mit dem Nullelement. Geben wir im Direktmodus A(0)=10:A(1)=11 ein, so erkennen wir die Fließkommawerte:

```
>M 0810 0820
>: 0810 41 00 D0 09 01 01 F5 84
>: 0818 20 00 00 00 84 30 00 00
>: 0820 00 00 00 00 00 00 00 00
```

```
84 20 00 00 00  => 10;  84 30 00 00 00  => 11
```

Wie sieht die Speicherbelegung bei mehrdimensionalen Arrays aus? Dazu löschen wir das Programm und geben im Direktmodus

DIM B(1,2,3)

ein. Unsere Arraytabelle beginnt bei $0803 und sieht so aus:

```
>M 002B 0037
>: 002B 01 08 03 08 03 08 86 08
>: 0033 00 A0 00 00 00 A0

>M 0803 0813
>: 0803 42 00 83 00 03 00 04 00
>: 080B 03 00 02 00 00 00 00 00
>: 0813 00 00 00 00 00 00 00 00
```

Wir erkennen wieder den Namen "B" gleich $42. Die Länge der Arraytabelle ist diesmal $0083 = 131 Bytes. Dann folgt eine 3, die anzeigt, daß unser Array dreidimensional ist. Nun kommen die Indexgrenzen, und zwar beginnend mit dem letzten Index $0004, dann $0003 und $0002 entsprechend 3, 2 und 1. Wie werden die Werte abgelegt? Durch Experimentieren kann man diese Reihenfolge ermitteln:

B(0,0,0)
B(1,0,0)
B(0,1,0)
B(1,1,0)
B(0,2,0)
B(1,2,0)
B(0,0,1)
B(1,0,1)
B(0,1,1)
B(1,1,1)
B(0,2,1)
B(1,2,1)
B(0,0,2)

B(1,0,2)
B(0,1,2)
B(1,1,2)
B(0,2,2)
B(1,2,2)
B(0,0,3)
B(1,0,3)
B(0,1,3)
B(1,1,3)
B(0,2,3)
B(1,2,3)

Sie können erkennen, daß der am weitesten vorne stehende Index am häufigsten wechselt, der an weitesten hinten stehende an wenigsten.

Benutzen wir Arrays mit INTEGER-Variablen, so werden hier für jedes Arrayelement nur 2 Bytes reserviert, so daß sich hier eine Platzersparnis gegenüber Fließkommaarrays ergibt. Bei Stringarrays werden nur drei Bytes pro Element gebraucht, jeweils für Länge und Adresse des Strings. Dazu kommt natürlich noch der Platz für die Strings selber. Mit diesen Erkenntnissen können wir den Platzbedarf eines beliebigen Arrays ermitteln:

$$P = 5 + 2*N + T * PROD(N_I+1)$$

Dabei ist P der erforderliche Platzbedarf des gesamten Arrays, N die Anzahl der Dimensionen, T der spezifische Platzbedarf eines Elements (2 für INTEGER, 5 für REAL und 3 für STRING) und $PROD(N_I+1)$ das Produkt der Indexgrenzen + 1.

Die Formel erklärt sich folgendermaßen:

Die Konstante 5 setzt sich zusammen aus 2 Bytes für den Namen, 2 Bytes für die Länge und ein Byte für die Anzahl der Dimensionen. Dann werden für jede Dimension zwei Bytes für die Indexgrenzen benötigt. Der Platz für die Elemente selbst ist im letzten Term enthalten. Probieren wir unsere Formel mit dem ersten Array A(500) aus.

P = 5 + 2*1 + 5*(501)
P = 2512 Bytes

Unser dreidimensionales Array B(1,2,3) hat dann folgenden Speicherplatzbedarf:

P = 5 + 2*3 + 5*(2*3*4)
P = 131 Bytes

Ein Array vom Typ A%(10,10,10) benötigt diesen Speicherplatz.

P = 5 + 2*3 + 2*(11*11*11)
P = 2673 Bytes

Ein Stringarray A$(100,100) würde kaum in den Speicher passen

P = 5 + 2*2 + 3*(101*101)
P = 30603

Hier würde die Arraytabelle also 30 KByte beanspruchen; für die 10201 Elemente selbst stünden nur noch 8 KByte zur Verfügung.

1.5 Fließkommafunktionen des BASIC-Interpreters

Nachdem wir bereits wissen, wie man die Grundrechenarten in Fließkommaarithmetik durchführt, sind jetzt die Funktionen an der Reihe.

Eine Funktion kann allgemein so beschrieben werden:

Y = F(X)

Dabei ist X das Argument, F die Funktion und Y das Ergebnis. Die Fließkommafunktionen sind so geschrieben, daß das Argument X im FAC bereit stehen muß, bevor die Funktion aufgerufen werden kann. Das Ergebnis des Funktionsaufrufs steht anschließend wieder im FAC zur Verfügung.

Der BASIC-Interpreter stellt uns eine ganze Reihe Funktionen zur Verfügung, die wir Ihnen nun vorstellen möchten.

Name	Adresse	Rechenzeit	Beschreibung
ABS	$BC58	0.0 ms	Absolutwert
ATN	$E30E	44.6 ms	Arcus tangens
COS	$E264	27.9 ms	Cosinus
EXP	$BFED	26.6 ms	Potenz zur Basis e
FRE	$B37D	0.6 ms	Freier Speicherplatz
INT	$BCCC	0.9 ms	Ganzzahliger Anteil
LOG	$B9EA	22.2 ms	Natürlicher Logarithmus
POS	$B39E	0.3 ms	Cursorspalte
RND	$E097	3.5 ms	Zufallszahl
SGN	$BC39	0.4 ms	Vorzeichen
SIN	$E26B	24.5 ms	Sinus
SQR	$BF71	51.2 ms	Quadratwurzel
TAN	$E2B4	49.8 ms	Tangens

Die Rechenzeiten wurden mit Pi als Argument ermittelt. Wie Sie anhand der Tabelle sehen können, unterscheiden sie sich doch gewaltig. Vor allem die sogenannten transzendeten Funktionen wie COS, EXP, LOG, SIN, TAN und ATN benötigen relativ viel Zeit. Diese Funktionen können durch die vier Grundrechenarten nicht mehr exakt berechnet werden. Man arbeitet dabei mit Näherungsverfahren, die eine endliche Genauigkeit ergeben. Die meisten Funktionen werden dabei durch Polynome angenähert, das sind Funktionen der Form

$$y = a_0 + a_1*x + a_2*x^2 + a_3*x^3 + a_4*x^4 + a_5*x^5 + ...$$

Je mehr Glieder eine solche Reihe hat, desto genauer wird das Ergebnis, desto länger dauert aber auch die Berechnung.

Wollte man solch ein Polynom so berechnen, wie es geschrieben wird, so wären z.B. bei dem dargestellten Polynom 5. Grades

1 + 2 + 3 + 4 + 5 = 15 Multiplikationen
und 5 Additionen erforderlich.

Aus der Mathematik kennt man ein rationelleres Verfahren, das unter dem Namen 'Horner-Schema' bekannt ist. Dazu formuliert man die obige Gleichung um.

$$y = ((((a_5 * x + a_4) * x + a_3) * x + a_2) * x + a_1) * x + a_0$$

Die Klammern sind dabei nur erforderlich, wenn man Punkt- vor Strichrechnung gehen läßt. Hierbei sind also nur noch 5 Multiplikationen und 5 Additionen erforderlich, allgemein bei einen Polynom N. Grades N Multiplikationen und N Additionen gegenüber N*(N-1)/2 Multiplikationen und N Additionen.

Die Einfachheit dieses Verfahrens kann man an einem entsprechenden BASIC-Programm zeigen.

```
100 Y = A(N)
110 FOR I = N-1 TO 0 STEP -1
120 Y = Y * X + A(I)
130 NEXT
```

Das Programm berechnet den Wert des Polynoms Nten Grades für den Wert X und gibt das Ergebnis in Y zurück. Das Feld A(0) bis A(N) enthält die Koeffizienten a_0 bis a_N.

Diese Routine zur Polynomauswertung ist das Kernstück sämtlicher transzendenter Funktionen, die der BASIC-Interpreter berechnen muß.

Wollen wir diese Routine benutzen, so muß das Argument, für das wir den Polynomwert berechnen wollen, im FAC bereit gestellt werden. Die Polynomkoeffizienten müssen in folgendem Format im Speicher stehen:

Polynomgrad n
Koeffizient n. Grades
Koeffizient n-1 . Grades
....
Koeffizient 1. Grades
Koeffizient 0. Grades

Der Polynomgrad ist als Ein-Byte-Wert gespeichert, die Koeffizienten müssen als 5-Byte Fließkommazahlen folgen. Beim Aufruf muß die Adresse dieses Koeffizientenfeldes übergeben werden. Im Akku muß dazu das Lo-Byte stehen, im Y-Register das Hi-Byte. Mit diesen Kenntnissen können wir schon eine Routine schreiben, die ein beliebiges Polynom berechnet.

Mit einen normalen Assembler ist es relativ umständlich, Fließkommawerte im Objektkode abzulegen. Wir können dabei so vorgehen, daß wir einer Variablen die Werte zuweisen, mit Hilfe des Monitors die Variablentabelle suchen, die entsprechenden 5 Bytes des Variablenwerts notieren und diese dann mittels des .BYT-Befehls in den Quelltext einfügen. Mit PROFI-MAT 2.0 haben Sie jedoch die Möglichkeit, Fließkommakonstanten direkt einzusetzen. Dies geschieht mit dem .FLP-Befehl (floating point). Der Assembler übernimmt dann die Umwandlung in die interne 5-Byte-Darstellung.

Versuchen wir einmal, unsere Kenntnisse in die Praxis umzusetzen und berechnen folgendes Polynom:

$$y = 0.7 + 2.5 * x + 8.2 * x^2 - 2.3 * x^3 + 0.5 x^4$$

```
PROFI-ASS 64 V2.0     SEITE 1

100:  033C                      .OPT P,00
110:                 ;
120:                 ; POLYNOMBERECHNUNG
130:                 ;
140:  033C                      *=   828      ; KASSETTENPUFFER
150:                 ;
160:  E059           POLYNOM    =    $E059
170:                 ;
180:  033C A9 43                LDA  #< KOEFF
190:  033E A0 03                LDY  #> KOEFF
200:  0340 4C 59 E0             JMP  POLYNOM
210:                 ;
220:  0343 04        KOEFF      .BYT 4        ; POLYNOMGRAD
230:  0344 80 00 00             .FLP 0.5      ; A(4)
240:  0349 82 93 33             .FLP -2.3     ; A(3)
```

```
250:    034E 84 03 33          .FLP 8.2    ; A(2)
260:    0353 82 20 00          .FLP 2.5    ; A(1)
270:    0358 80 33 33          .FLP 0.7    ; A(0)
280:                 ;
]033C-035D
NO ERRORS
```

Die ganze Routine beschränkt sich also auf die Übergabe der Startadresse und den Aufruf der Polynomfunktion; es folgen dann die Koeffizienten des Polynoms in absteigender Reihenfolge.

Doch wie können wir unsere neue Funktion anwenden? Mit dem SYS-Befehl geht es offensichtlich nicht - wie sollten wir das Funktionsargument übergeben und wie den Funktionswert zurückerhalten? Wir brauchen eine Funktion ähnlich den eingebauten Funktionen wie SIN, EXP usw.

Dieser Fall wurde bereits im Interpreter berücksichtigt. Es gibt dazu die USR-Funktion, die der Anwender (User) frei definieren kann. Wir brauchen dazu dem Interpreter nur mitzuteilen, an welcher Adresse unsere eigene Funktion startet. Diese Startadresse wird in der üblichen Form Lo/Hi-Byte an den Adressen 785/786 ($0311/$0312) hinterlegt und schon können wir unsere neue Funktion benutzen.

POKE 785,828 AND 255 : POKE 786,828/256

Geben Sie jetzt, nachdem Sie das obige Programm assembliert haben, einmal ein:

? USR(1)

Sie erhalten den Wert 9.6. Eine Überprüfung obiger Formel

bestätigt die Richtigkeit des Ergebnisses.

y = 0.7 + 2.5 + 8.2 - 2.3 + 0.5 = 9.6

Zur weiteren Kontrolle können Sie folgende Schleife eingeben.

```
FOR I=-5 TO 5 : PRINT USR(I) : NEXT

793.2
397.1
169.6
54.9
9.2
.7
9.6
28.1
60.4
122.7
243.2
```

Dieses Verfahren zur Berechnung von Polynomen empfiehlt sich immer dann, wenn ein Programm ein Polynom wiederholt berechnen muß. Die Ausführungzeit dieser Funktion ist mit 12.5 ms sogar noch kürzer als viele andere eingebaute Funktionen. Die Berechnung in BASIC braucht ca. 45 ms. Je komplizierter die Formel wird, um so schneller ist im Vergleich die Maschinenspracheversion.

Wie Sie aus dem obigen Beispiel entnehmen konnten, müssen die Koeffizienten einschließlich ihrer Vorzeichen in absteigender Reihenfolge (d.h. mit dem Koeffizienten der höchsten Potenz beginnend) verwendet werden. Fehlt eine Potenz von x im Polynom, so ist anstelle dessen der Wert null einzusetzen.

Das nächste Beispiel soll die Fakultät-Funktion berechnen. Die Fakultät ist eine Funktion, die zunächst nur für ganzzahlige positive Werte definiert ist und aus dem Produkt aller Zahlen von eins bis zum berechnenden Wert besteht, z.B.

5! = 1 * 2 * 3 * 4 * 5 = 120

oder

7! = 1 * 2 * 3 * 4 * 5 * 6 * 7 = 5040

In der Mathematik hat man die Definition der Fakultät auch auf nichtganzzahlige Werte ausgedehnt, die wiederum durch ein Polynom näherungsweise berechnet werden können. Dieses Polynom ist jedoch nur für Werte zwischen 0 und eins definiert; Funktionswerte von anderen Argumenten müssen dabei zurückgerechnet werden, z.B.

4.3! = 4.3 * 3.3 * 2.3 * 1.3 * 0.3!

Die Fakultät von 0.3 kann mit einem Polynom achten Grades berechnet werden, das folgende Koeffzienten hat:

a_0 = 1
a_1 = -.57719 1652
a_2 = .98820 6891
a_3 = -.89705 6937
a_4 = .91820 6857
a_5 = -.75670 4078
a_6 = .48219 9394
a_7 = -.19352 7818
a_8 = .03586 8343

Diese Polynomfunktion können wir nun programmieren.

```
PROFI-ASS 64 V2.0    SEITE 1

100:   033C                      .OPT P1,00
110:                   ;
120:                   ; POLYNOM ZUR FAKULTAET-BERECHNUNG
130:                   ;
140:   033C                      *=   828       ; KASSETTENPUFFER
150:                   ;
160:   E059            POLYNOM   =    $E059
170:                   ;
180:   033C A9 43                LDA  #< KOEFF
190:   033E A0 03                LDY  #> KOEFF
200:   0340 4C 59 E0             JMP  POLYNOM
210:                   ;
220:   0343 08         KOEFF     .BYT 8         ; POLYNOM 8.GRADES
230:   0344 7C 12 EA             .FLP .035868343
240:   0349 7E C6 2C             .FLP -.193527818
250:   034E 7F 76 E2             .FLP .482199394
260:   0353 80 C1 B7             .FLP -.756704078
270:   0358 80 6B 0F             .FLP .918206857
280:   035D 80 E5 A5             .FLP -.897056937
290:   0362 80 7C FB             .FLP .988206891
300:   0367 80 93 C2             .FLP -.577191652
310:   036C 81 00 00             .FLP 1
]033C-0371
NO ERRORS
```

Mit PRINT USR(X) können wir schon die Fakultätwerte für Argumente zwischen 0 und 1 berechnen, z.B.

```
?USR(.1) => 0.951350564
?USR(.5) => 0.886227246
```

Mit einer kleinen BASIC-Routine können wir auch die Fakultätwerte für Zahlen außerhalb dieses Bereichs berechnen.

```
10 INPUT "ARGUMENT"; X
20 IF X<0 OR X>33 THEN 10
30 IF X=0 THEN Y=1 : GOTO 70
40 Y=X : IF X<1 THEN Y = USR(X) :GOTO70
50 X=X-1 : IF X>1 THEN Y=Y*X : GOTO 50
60 IF X<>1 THEN Y = Y * USR(X)
70 PRINT "FAKULTAET = "; Y
```

In Zeile 20 werden negative Werte, sowie Werte, deren Fakultät größer als 1E38 ist, abgefangen. Der Wert null ergibt definitionsgemäß eins (Zeile 30). In Zeile 50 wird solange multipliziert und um eins vermindert, bis der Wert kleiner oder gleich eins ist. In Zeile 60 wird geprüft, ob es sich um einen ganzzahligen Wert handelt. Ist dies nicht der Fall, so muß noch mit dem Polynomwert multipliziert werden, und in Zeile 70 kann dann das Ergebnis ausgegeben werden, z.B.

0 => 1
1 => 1
1.5 => 1.32934087
2 => 2
3 => 3
0.5 => .886227246
7.35 => 10287.3151

Nachdem wir bereits das Polynom mit einer eigenen Maschinenspracheroutine berechnet haben, wollen wir versuchen, ob wir nicht das komplette BASIC-Programm durch ein Maschinenprogramm ersetzen können. Dabei werden wir weitere Routinen der Fließkommaarithmetik kennenlernen.

Dazu entwerfen wir ein Ablaufdiagramm.

START

FAC nach TEMP

FAC negativ? — Y — ILLEGAL QUANTITY — 1

FAC >34 ? — Y — OVERFLOW ERROR — 1

FAC = 0 ? — Y — FAC = 1 — 1

FAC < 1 ? — Y — FAC = POLY(FAC) — 1

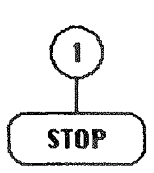
FAC = FAC - 1

FAC > 1 ? — Y — TEMP = TEMP * FAC

FAC = 1 ? — Y — FAC = TEMP — 1

FAC = POLY(FAC)
FAC = FAC * TEMP

1



```
100:   033C                      .OPT P1,00
110:   033C           START    =    828      ; Kassettenpuffer
120:   B97E           OVERFLOW =    $B97E    ; OVERFLOW ERROR
130:   B248           ILLQUAN  =    $B248    ; ILLEGAL QUANTITY
140:                  ;
150:   BBD4           FACMEM   =    $BBD4    ; FAC abspeichern
160:   BBA2           MEMFAC   =    $BBA2    ; FAC laden
170:   BC5B           VERGLCH  =    $BC5B    ; Konstante mit FAC vergleichen
180:   B867           MEMPLUS  =    $B867    ; Konstante plus FAC
190:   BA28           MEMMULT  =    $BA28    ; Konstante mal FAC
200:   E059           POLYNOM  =    $E059    ; Polynomberechnung
210:   0066           SIGN     =    $66      ; Vorzeichen
220:   0061           EXP      =    $61      ; Exponent
230:                  ;
240:   033C                    *=   START
250:   033C A2 F5              LDX  #< TEMP
260:   033E A0 03              LDY  #> TEMP
270:   0340 20 D4 BB           JSR  FACMEM   ; FAC nach TEMP speichern
280:                  ;
290:   0343 24 66              BIT  SIGN     ; Vorzeichen testen
300:   0345 10 03              BPL  OK1      ; positiv ?
310:   0347 4C 48 B2           JMP  ILLQUAN
320:   034A A9 F0     OK1      LDA  #< MAX
330:   034C A0 03              LDY  #> MAX   ; Zeiger auf Konstante 34
340:   034E 20 5B BC           JSR  VERGLCH
350:   0351 30 03              BMI  OK2      ; kleiner ?
360:   0353 4C 7E B9           JMP  OVERFLOW
370:   0356 A5 61     OK2      LDA  EXP      ; FAC = 0
380:   0358 F0 52              BEQ  GLCHEINS  ; dann 1
390:   035A A9 E6              LDA  #< EINS
400:   035C A0 03              LDY  #> EINS
410:   035E 20 5B BC           JSR  VERGLCH   ; mit eins vergleichen
```

```
420:   0361 30 50            BMI  KLEINER
430:   0363 A9 EB   SCHLEIFE LDA  #< MINUS1
440:   0365 A0 03            LDY  #> MINUS1
450:   0367 20 67 B8         JSR  MEMPLUS
460:   036A A9 E6            LDA  #< EINS
470:   036C A0 03            LDY  #> EINS
480:   036E 20 5B BC         JSR  VERGLCH  ; mit eins vergleichen
490:   0371 30 1F            BMI  NEXT    ; nicht mehr größer ?
500:   0373 A2 FA            LDX  #< TEMP2
510:   0375 A0 03            LDY  #> TEMP2
520:   0377 20 D4 BB         JSR  FACMEM  ; FAC zwischenspeichern
530:   037A A9 F5            LDA  #< TEMP
540:   037C A0 03            LDY  #> TEMP
550:   037E 20 28 BA         JSR  MEMMULT  ; FAC * TEMP
560:   0381 A2 F5            LDX  #< TEMP
570:   0383 A0 03            LDY  #> TEMP
580:   0385 20 D4 BB         JSR  FACMEM  ; Ergebnis wieder nach TEMP
590:   0388 A9 FA            LDA  #< TEMP2
600:   038A A0 03            LDY  #> TEMP2
610:   038C 20 A2 BB         JSR  MEMFAC  ; FAC wiederholen
620:   038F 4C 63 03         JMP  SCHLEIFE
630:                ;
640:   0392 A9 E6   NEXT     LDA  #< EINS
650:   0394 A0 03            LDY  #> EINS
660:   0396 20 5B BC         JSR  VERGLCH  ; ist FAC = 1 ?
670:   0399 00 07            BNE  WEITER
680:                ;
690:   039B A9 F5            LDA  #< TEMP
700:   0390 A0 03            LDY  #> TEMP
710:   039F 4C A2 BB         JMP  MEMFAC  ; nach FAC holen
720:                ;
730:   03A2 20 B6 03 WEITER  JSR  POLY    ; Polynom berechnen
```

```
PROFI-ASS 64 V2.0    SEITE 3

03A5 A9 F5              LDA  #< TEMP
03A7 A0 03              LDY  #> TEMP
03A9 4C 28 BA           JMP  MEMMULT
                ;
03AC A9 E6      GLCHEINS LDA #< EINS
03AE A0 03              LDY  #> EINS
03B0 4C A2 BB           JMP  MEMFAC   ; 1 in FAC
                ;
03B3 4C B6 03 KLEINER  JMP  POLY     ; FAC = POLY(FAC)
                ;
                ; Polynom zur Fakultät-Berechnung
                ;
03B6 A9 BD      POLY    LDA  #< KOEFF
03B8 A0 03              LDY  #> KOEFF
03BA 4C 59 E0           JMP  POLYNOM
                ;
03BD 08         KOEFF   .BYT 8        ; Polynom 8.Grades
03BE 7C 12 EA           .FLP .035868343
03C3 7E C6 2C           .FLP -.193527818
03C8 7F 76 E2           .FLP .482199394
03CD 80 C1 B7           .FLP -.756704078
03D2 80 6B OF           .FLP .918206857
03D7 80 E5 A5           .FLP -.897056937
03DC 80 7C FB           .FLP .988206891
03E1 80 93 C2           .FLP -.577191652
                ;
                ; weitere Fließkommakonstanten
03E6 81 00 00 EINS      .FLP 1
03EB 81 80 00 MINUS1    .FLP -1
03F0 86 08 00 MAX       .FLP 34
                ;
```

```
1050:                 ; Speicherplatz für temporäre Variable
1060:  03F5           TEMP     *=    *+5
1070:  03FA           TEMP2    *=    *+5
]033C-03FF
NO ERRORS
```

Probieren wir unsere neue Funktion direkt einmal aus! (Vergessen Sie vorher jedoch nicht, den USR-Vektor an Adresse 785/786 auf unsere Routine zu setzen - nach dem Einschalten des Rechners zeigt dieser Vektor immer auf 'ILLEGAL QUANTITY'.)

```
?USR(0)   =>  1
?USR(1)   =>  1
?USR(2)   =>  2
?USR(3)   =>  6
?USR(.5)  =>  .886227246
?USR(4.5) =>  52.3427967
?USR(-1)  =>  ILLEGAL QUANTITY ERROR
?USR(40)  =>  OVERFLOW ERROR
```

Was wir vorher noch mit einem relativ umständlichen BASIC-Programm erledigen mußten, geschieht nun bedeutend schneller und komfortabler durch den Aufruf einer einzigen Funktion. In dem Maschinenprogramm haben wir einige neue Routinen verwendet, die wir kurz besprechen wollen.

FACMEM - Diese Routine speichert den Inhalt des Fließkommaakkus FAC an die Adresse, die in X (lo-Byte) und Y (hi-Byte) angegeben wird. Der Inhalt des FAC wird in der verkürzten 5-Byte-Form abgelegt.

MEMFAC - Damit wird die umgekehrte Aufgabe erledigt. Es wird eine Fließkommazahl aus dem Speicher in den FAC geholt. Diesmal müssen A (lo-Byte) und Y (hi-Byte) die Speicheradresse enthalten.

VERGLCH - Mit diesem Unterprogramm des BASIC-Interpreters können wir zwei Fließkommazahlen mit einander vergleichen. Die erste Zahl steht dabei im Speicher und wird wieder durch A (lo-Byte) und Y (hi-Byte) adressiert. Die zweite Zahl muß sich im FAC befinden. Sind beide Zahlen gleich, so enthält der Akku (nicht der Fließkommaakku!) eine Null und das Z-Flag ist gesetzt. Ist der erste Wert kleiner als die Zahl im FAC, so enhält der Akku -1 ($FF) und das N-Flag ist gesetzt. War dagegen die Zahl im FAC größer, enhält der Akku eins und das N-Flag ist gelöscht. Diese Routine haben wir in unserem Programm ausgiebig angewendet.

MEMPLUS - Diese Routine faßt zwei Unterprogramme zusammen. Zum ersten wird die Fließkommazahl, auf die A und Y zeigen (lo/hi), nach ARG geholt und anschließend wird die Routine zur Addition von FAC und ARG aufgerufen, die das Ergebnis im FAC läßt.

MEMMULT - Zur Multiplikation einer Zahl im Speicher mit dem FAC dient diese Routine. Die Logik entspricht dabei der von MEMPLUS.

Die Adressen **OVERFLOW** und **ILLQUAN** rufen die entsprechenden Routinen zur Ausgabe der Fehlermeldungen auf. Genau genommen war es in unserem Falle sogar unnötig, das Funktionsargument auf größer 34 zu prüfen, da im Laufe der Multiplikationen diese Fehlermeldung automatisch erschienen wäre.

Die Funktion zur Polynomberechnung gibt es noch in einer abgewandelten Form, die folgende Formel entwickelt:

$$y = a_0 * x + a_1 * x^3 + a_2 * x^5 + a_3 * x^7 + \ldots$$

Diese Funktion leitet sich aus der normalen Polynomberechnung dadurch ab, daß man als Argument x_2 nimmt und das Ergebnis nochmal mit x multipliziert.

$$y = x *(a_0 + a_1 * (x^2) + a_2 * (x^2)^2 + a_3 * (x^2)^3 + \ldots)$$

Diese Routine wird bei den meisten eingebauten Funktionen benutzt, da das Näherungspolynom oft von dieser Form ist. Vorher werden die Argumente meist noch in einen bestimmten Wertebereich gebracht, für den diese Funktion nur definiert ist und anschließend wird das Ergebnis entsprechend dem Ursprungswert noch modifiziert.

Berechnen wir mit dieser Routine folgende Formel:

$$y = 6 * x + 0.5 * x^3 - 0.11 * x^7$$

Beachten Sie, daß hier ein Glied der Reihe fehlt (das mit dem Exponenten 5), wir müssen hierfür Null als Koeffizient einsetzen.

```
PROFI-ASS 64 V2.0    SEITE 1

100:   033C                 .OPT P,00
110:   033C                 *=   828
120:               ;
130:   E043        POLY2    =    $E043
140:               ;
150:   033C A9 43           LDA  #< KOEFF
160:   033E A0 03           LDY  #> KOEFF
170:   0340 4C 43 E0        JMP  POLY2
180:               ;
190:   0343 03     KOEFF    .BYT 3        ; Polynomgrad
200:   0344 7D E1 47        .FLP -.11
210:   0349 00 00 00        .FLP 0
220:   034E 80 00 00        .FLP .5
230:   0353 83 40 00        .FLP 6
]033C-0358
NO ERRORS
```

Beachten Sie hierbei, daß der Polynomgrad sich aus der Nummer des höchsten Koeffizienten ergibt, nicht aus der höchsten Potenz, da wir ja x ausgeklammert haben und als Argument x_2 verwenden.

Zur Kontrolle hier ein paar Funktionswerte:

```
USR(0)   = 0
USR(1)   = 6.39
USR(2)   = 1.92
USR(.75) = 4.69625427
```

Zum Abschluß unserer Betrachtung über die Fließkommazahlen wollen wir noch ein Problem aufgreifen, das bei der Programmierung häufig auftritt: Das Sortieren von Zahlenfeldern. Wir versuchen den folgenden Algorithmus in Maschinensprache zu implementieren.

```
100 FOR I=1 TO N : FL=0
110 FOR J=N TO I STEP -1
120 IF A(J-1)>A(J) THEN H=A(J):A(J)=A(J-1):A(J-1)=H : FL=1
130 NEXT J
140 IF FL=0 THEN RETURN
150 NEXT I : RETURN
```

Das Programm sortiert das Feld A(N) und kann als Unterprogramm mit GOSUB 100 aufgerufen werden. Das Programm arbeitet nach dem sogenannten Bubble-Sort Algorithmus. Dabei werden je zwei nebeneinanderstehende Feldelemente verglichen. Ist das zuerst stehende Element größer als das darauf folgende, so werden die beiden Elemente ausgetauscht und ein Flag gesetzt. Dies geschieht in zwei geschachelten Schleifen. War innerhalb eines inneren Schleifendurchlaufs kein Tausch erforderlich, so ist das Feld schon sortiert. In diesem Falle bleibt das Flag auf null und der Sortiervorgang kann vorzeitig abgebrochen werden. Ansonsten befindet sich nach dem ersten Durchlauf der kleinste Wert im ersten Element A(0). Nun wird der gleiche Durchlauf für die Elemente 1 bis N, dann 2 bis N usw. durchgeführt. Wissen Sie noch, wie in BASIC Feldelemente gespeichert werden? Nun - es existiert ein Zeiger, der den Beginn der Arraytabelle kennzeichnet. Damit wir diese Tabelle nicht erst nach dem richtigen Feld durchsuchen müssen, einigen wir uns darauf, daß unser zu sortierendes Array als erstes dimensioniert werden muß, so daß es auch als erstes in der Tabelle steht.

```
PROFI-ASS 64 V2.0    SEITE 1

100:   033C                       .OPT P,00
110:   002F          ARRTAB       =    $2F      ; Zeiger auf Arraytabelle
120:   0057                       *=   $57
130:   0057          IPNT         *=   *+2
140:   0059          JPNT         *=   *+2
150:   005B          JPNT1        *=   *+2
160:                 ;
170:   SBA2          MEMFAC       =    $BBA2
180:   BC5B          VERGLCH      =    $BC5B
190:                 ;
200:   033C                       *=   828      ; Kassettenpuffer
210:                 ;
220:   033C A5 2F                 LDA  ARRTAB
230:   033E 18                    CLC
240:   033F A0 02                 LDY  #2
250:   0341 71 2F                 ADC  (ARRTAB),Y  ; Arraylänge addieren
260:   0343 8D D9 03              STA  NPNT     ; Zeiger N auf Arrayende
270:   0346 C8                    INY
280:   0347 A5 30                 LDA  ARRTAB+1
290:   0349 71 2F                 ADC  (ARRTAB),Y
300:   034B 8D DA 03              STA  NPNT+1
310:   034E AD D9 03              LDA  NPNT
320:   0351 38                    SEC
320:   0352 E9 05                 SBC  #5
330:   0354 8D D9 03              STA  NPNT
340:   0357 B0 03                 BCS  L1
350:   0359 CE DA 03              DEC  NPNT+1
360:                 ;
370:   035C A5 2F    L1           LDA  ARRTAB
380:   035E 18                    CLC
390:   035F 69 07                 ADC  #7
400:   0361 85 57                 STA  IPNT     ; Zeiger I auf A(0)
```

```
410:   0363 A5 30              LDA  ARRTAB+1
420:   0365 69 00              ADC  #0
430:   0367 85 58              STA  IPNT+1
440:                  ;
450:   0369 A0 00     ILOOP    LDY  #0
460:   036B 8C D8 03           STY  FLAG     ; Flag löschen
470:   036E AD D9 03           LDA  NPNT
470:   0371 85 59              STA  JPNT     ; J=N
480:   0373 AD DA 03           LDA  NPNT+1
480:   0376 85 5A              STA  JPNT+1
490:                  ;
500:   0378 A5 59     JLOOP    LDA  JPNT
510:   037A 38                 SEC
510:   037B E9 05              SBC  #5
520:   037D 85 5B              STA  JPNT1    ; Zeiger J-1
530:   037F AA                 TAX
540:   0380 A5 5A              LDA  JPNT+1
550:   0382 E9 00              SBC  #0
560:   0384 85 5C              STA  JPNT1+1
560:   0386 A8                 TAY
560:   0387 8A                 TXA
570:   0388 20 A2 BB           JSR  MEMFAC   ; A(J-1) nach FAC
580:                  ;
590:   038B A5 59              LDA  JPNT
600:   038D A4 5A              LDY  JPNT+1
610:   038F 20 5B BC           JSR  VERGLCH ; mit A(J) vergleichen
620:   0392 30 12              BMI  NOTAUSCH
630:                  ;
640:   0394 A0 04              LDY  #4
640:   0396 8C D8 03           STY  FLAG     ; Flag setzen
650:   0399 B1 59     SWAP     LDA  (JPNT),Y
660:   039B AA                 TAX
```

```
670:   039C B1 5B              LDA  (JPNT1),Y
680:   039E 91 59              STA  (JPNT),Y ; A(J) und A(J-1)
690:   03A0 8A                 TXA            ; tauschen
700:   03A1 91 5B              STA  (JPNT1),Y
710:   03A3 88                 DEY
720:   03A4 10 F3              BPL  SWAP
730:                  ;
740:   03A6 A5 59     NOTAUSCH LDA  JPNT
750:   03A8 38                 SEC
750:   03A9 E9 05              SBC  #5        ; J=J-1
760:   03AB 85 59              STA  JPNT
770:   03AD B0 02              BCS  TESTJ
780:   03AF C6 5A              DEC  JPNT+1
790:                  ;
800:   03B1 C5 57     TESTJ    CMP  IPNT
810:   03B3 D0 C3              BNE  JLOOP
820:   03B5 A5 5A              LDA  JPNT+1    ; I=J ?
830:   03B7 C5 58              CMP  IPNT+1
840:   03B9 D0 BD              BNE  JLOOP
850:                  ;
860:   03BB AD D8 03           LDA  FLAG      ; hat kein Tausch
870:   03BE F0 17              BEQ  ENDE      ; stattgefunden ?
880:                  ;
890:   03C0 A5 57              LDA  IPNT
900:   03C2 18                 CLC
900:   03C3 69 05              ADC  #5        ; I=I+1
910:   03C5 85 57              STA  IPNT
920:   03C7 90 02              BCC  TESTI
930:   03C9 E6 58              INC  IPNT+1
940:                  ;
950:   03CB CD D9 03  TESTI    CMP  NPNT
960:   03CE D0 99              BNE  ILOOP
```

```
PROFI-ASS 64 V2.0    SEITE 4

970:   03D0 A5 58              LDA  IPNT+1  ; I=N ?
980:   03D2 CD DA 03           CMP  NPNT+1
990:   03D5 D0 92              BNE  ILOOP
1000:                ;
1010:  03D7 60       ENDE      RTS
1020:                ;
1030:  03D8          FLAG      *=   *+1
1040:  03D9          NPNT      *=   *+2
]033C-03DB
NO ERRORS
```

Das Programm übernimmt die Aufgabe des obigen BASIC-Unterprogramms. Dazu muß, wie gesagt, das zu sortierende Array als erstes dimensioniert sein. Das Programm prüft nicht, ob das Array überhaupt angelegt wurde und ob es eindimensional ist, das liegt in der Verantwortung des Benutzers.

Soll ein Array sortiert werden, so genügt einfach der Aufruf der Routine mit

SYS 828

Um eine ungefähre Vorstellung von der Geschwindigkeit des Programms zu bekommen, haben wir verschieden große Arrays mit Zufallszahlen einmal in BASIC, zum anderen mit der Maschinenroutine sortiert. Die Ergebnisse finden Sie in der folgenden Tabelle.

N	BASIC	MASCHINENROUTINE
10	1 "	0.0 "
50	24 "	0.4 "
100	1' 37 "	1.5 "
200	6' 33 "	6.3 "
500	41'	38.7 "
1000	2h 44'	2' 33.4 "

Aus der Tabelle kann man gut erkennen, wie die Sortierzeit etwa quadratisch ansteigt, d.h. für doppelt soviel Elemente benötigt man ca. viermal soviel Zeit. Sobald Sie größere Arrays sortieren müssen, kommt bald der Punkt, bei dem Zeitbedarf in BASIC im Stundenbereich liegt. Hier ist unsere Maschinenroutine im Schnitt gut sechzigmal schneller. Haben Sie sehr große Arrays und ist Ihnen die Maschinenroutine immer noch zu lang, so müssen Sie zu aufwendigeren Algorithmen übergehen, deren Zeitverhalten günstiger ist, z.B. Quicksort.

Versuchen Sie doch zur Übung einmal, unsere Routine dahingehend zu modifizieren, daß Sie INTEGER-Arrays sortieren können. Was muß dazu geändert werden? Zum einen muß der veränderte Platzbedarf eines Elements berücksichtigt werden - es müssen jeweils 2 statt 5 Bytes addiert bzw. subtrahiert werden. Zum anderen können wir den Vergleich der Elemente selbst durchführen. Dadurch entfällt die Umwandlung ins Fließkommaformat und der aufwendigere Vergleich von Fließkommazahlen, wir brauchen also nur die Zwei-Byte-Werte direkt zu vergleichen. Dadurch wird die Routine außerdem noch schneller als die Fließkommasortierung.

Zum Nachschlagen für Ihre eigenen Anwendungen finden Sie nun eine Tabelle, die alle Funktionen und Operationen des BASIC-Interpreters enthält, die sich mit der Arithmetik befassen.

Name	Adresse	Zeiger auf Konstante	Vorbereitung	FAC	Funktion
MEMARG	$BA8C	A/Y	-	-	ARG := Konstante
FACARG	$BBFC	-	-	+	FAC := ARG
DIV	$BB12	-	A = EXP	+	FAC := ARG / FAC
MEMDIV	$BB0F	A/Y	-	+	FAC := Konst/FAC
MAL10	$BAE2	-	-	+	FAC := FAC * 10
OURCH10	$BAFE	-	-	+	FAC := FAC / 10
PLUS05	$B849	-	-	+	FAC := FAC + 0.5
MEMFAC	$BBA2	A/Y	-	+	FAC := Konstante
FACARG	$BC0C	-	-	-	ARG := FAC
FACMEM	$BBD4	X/Y	-	-	Konstante := FAC
MINUS	$B853	-	A = EXP	+	FAC := ARG - FAC
MEMMIN	$B850	A/Y	-	+	FAC := Konst/FAC
MULT	$BA2B	-	A = EXP	+	FAC := ARG * FAC
MEMMULT	$BA28	A/Y	-	+	FAC := Konst*FAC
PLUS	$B86A	-	A = EXP	+	FAC := ARG + FAC
MEMPLUS	$B867	A/Y	-	+	FAC := Konst+FAC
HOCH	$BF7B	-	A = EXP	+	FAC := ARG ^ FAC
HOCHMEM	$BF78	A/Y	-	+	FAC := ARG ^ Konst
POLY	$E059	A/Y	-	+	FAC := Polynom
POLY2	$E043	A/Y	-	+	FAC := Polynom2
OR	$AFE6	-	-	+	FAC := ARG OR FAC
AND	$AFE9	-	-	+	FAC := ARG AND FAC
NOT	$AED4	-	-	+	FAC := NOT FAC
VERGLCH	$BC5B	A/Y	-	-	FAC mit Konst. vergleichen
ROUND	$BC1B	-	-	+	FAC runden
CHGSGN	$BFB4	-	-	+	FAC := -FAC

Konversionen und Standardfunktionen sind dabei nicht mehr enthalten, da sie bereits an anderer Stelle aufgeführt sind. Zur besseren und einfacheren Anwendung sind die Übergabeparameter und die genaue Funktion der Routinen beschrieben.

Das '+' in der Spalte FAC bedeutet, daß der Inhalt des FAC verändert wird, bei '-' bleibt er gleich. Soll eine Operation ARG und FAC miteinander verknüpfen, so ist vor dem Aufruf der Akku mit dem Exponenten von FAC ($61) zu laden.

Bei den logischen Operationen AND, OR und NOT werden die Argumente zuerst in 16-Bit-Integerzahlen umgewandelt, dann die Operation bitweise durchgeführt und das Ergebnis nach der Rückwandlung in eine Fließkommazahl wieder im FAC abgelegt.

Der BASIC-Interpreter beinhaltet eine Reihe von Fließkommazahlen für eigene Anwendungen. Die folgende Tabelle ermöglicht die Nutzung dieser Konstanten.

Adresse	Konstante	Dezimaler Wert	Bedeutung
$AEA8	82 49 0F DA A1	3.14159265	PI
$B1A5	90 80 00 00 00	-32768	
$B9BC	81 00 00 00 00	1	
$B9C2	7F 5E 56 CB 79	.434255942	
$B9C7	80 13 9B 0B 64	.576584541	
$B9CC	80 76 38 93 16	.961800759	
$B9D1	82 38 AA 3B 20	2.88539007	
$B9D6	80 35 04 F3 34	.707106781	1/SQR(2)
$B9DB	81 35 04 F3 34	1.41421356	SQR(2)
$B9E0	80 80 00 00 00	-.5	
$B9E5	80 31 72 17 F8	.693147181	LOG(2)
$BAF9	84 20 00 00 00	10	
$BDB3	9B 3E BC 1F FD	99999999.9	
$BDB8	9E 6E 6B 27 FD	999999999	

$BDBD	9E 6E 68 28 00	1E9	
$BFBF	81 38 AA 3B 29	1.44269504	1/LOG(2)
$BFC5	71 34 58 3E 56	2.14987637E-5	
$BFCA	74 16 7E B3 1B	1.4352314E-4	
$BFCF	77 2F EE E3 85	1.34226348E-3	
$BFD4	7A 1D 84 1C 2A	9.614011701E-3	
$BFD9	7C 63 59 58 0A	.0555051269	
$BFDE	7E 75 FD E7 C6	.240226385	
$BFE3	80 31 72 18 10	.693147186	
$BFE8	81 00 00 00 00	1	
$E08D	98 35 44 7A 00	11879546	
$E092	68 28 B1 46 00	3.92767774E-4	
$E2E0	81 49 0F DA A2	1.57079633	PI / 2
$E2E5	83 49 0F DA A2	6.28318531	PI * 2
$E2EA	7F 00 00 00 00	.25	
$E2F0	84 E6 1A 2D 1B	-14.3813907	
$E2F5	86 28 07 FB F8	42.0077971	
$E2FA	87 99 68 89 01	-76.7041703	
$E2FF	87 23 35 DF E1	81.6052237	
$E304	86 A5 5D E7 28	-41.3147021	
$E309	83 49 0F DA A2	6.28318531	PI * 2
$E33F	76 B3 83 BD D3	-6.84793912E-4	
$E344	79 1E F4 A6 F5	4.85094216E-3	
$E349	7B 83 FC B0 10	-.0161117015	
$E34E	7C 0C 1F 67 CA	.034209638	
$E353	7C DE 53 CB C1	-.054279133	
$E358	70 14 64 70 4C	.0724571965	
$E35D	7D B7 EA 51 7A	-.0898019185	
$E362	70 63 30 88 7E	.110932413	
$E367	7E 92 44 99 3A	-.142839808	
$E36C	7E 4C CC 91 C7	.19999912	
$E371	7F AA AA AA 13	-.333333316	
$E376	81 00 00 00 00	1	

2.1 Interruptprogrammierung

Ein Gebiet, das von vielen Maschinenprogrammierern gemieden wird, ist die Programmierung von Interrupt-Routinen. Diesem Umstand wollen wir abhelfen, die Prinzipien aufzeigen und beweisen, daß die Abneigung dagegen ganz und gar unbegründet ist.

Wir werden Ihnen erklären, was ein Interrupt ist und welche Möglichkeiten dem Maschinenprogrammierer durch diese neue Technik eröffnet werden.

Das Wort Interrupt kommt aus dem Englischen und bedeutet schlicht und ergreifend 'Unterbrechung'. Was wird dabei unterbrochen? Nun - ganz einfach das laufende Maschinenprogramm. Diese Unterbrechung eines laufenden Programms wird hardwaremäßig erreicht und kann an jeder beliebigen Stelle eines Programms auftreten. Wer oder was kann ein Maschinenprogramm unterbrechen? Dazu müssen wir kurz den hardwaremäßigen Aufbau des Prozessors betrachten.

Der 6502 bzw. 6510 Mikroprozessor ist in einem 40poligen Gehäuse untergebracht, von denen 2 Pins die Bezeichnung

IRQ und NMI

haben. Das sind die Abkürzungen für

Interrupt Request = Interrupt Anforderung

und

Non maskable Interrupt = nicht maskierbarer Interrupt

Tritt von außen ein Impuls an einem dieser Pins auf, so spielt sich folgendes ab.

1. Impuls am Pin NMI

Der Prozessor führt den augenblicklichen Befehl zu Ende aus und macht dann folgendes:

Der augenblickliche Wert des Programmzählers wird auf dem Stack abgelegt (erst Hi-Byte, dann Lo-Byte). Anschließend wird der Prozessorstatus (die Flags) ebenfalls auf den Stack gelegt. Nun holt sich der Prozessor den Inhalt der Adressen $FFFA und $FFFB und interpretiert sie als neuen Wert des Programmzählers, er führt also einen indirekten Sprung aus: JMP ($FFFA). Das ab dieser Adresse stehende Programm wird dann ausgeführt.

Dieses Programm 'bedient' die Interruptanforderung, doch dazu später.

2. Impuls am Pin IRQ

Hier passiert etwas ähnliches wie bei einem NMI. Der augenblickliche Befehl wird zu Ende ausgeführt, ehe auf den Interrupt reagiert wird. Bei einem IRQ prüft der Prozessor jedoch das Interrupt-Flag (Bit 3 im Statusregister). Ist dieses Flag gesetzt, so ignoriert er die Interruptanforderung und fährt mit der Abarbeitung des laufenden Programms fort. War das Flag jedoch nicht gesetzt, so wird analog wie beim NMI verfahren: Der Inhalt des Programmzählers und die Flags werden auf dem Stack abgelegt. Nun wird das I-Flag gesetzt, so daß eine weitere Interrupt-Anforderung innerhalb der Interruptroutine ignoriert wird. Den neuen Wert des Programmzählers holt der Prozessor sich von der Adresse $FFFE

und $FFFF. Der Wert, auf den diese Adresse zeigt, wird als neuer Wert des Programmzählers genommen.

Wie kann in das unterbrochene Programm zurückgekehrt werden? Dazu gibt es den speziellen Befehl

RTI - Return form Interrupt

Dieser Befehl macht das Umgekehrte wie beim Auslösen des Interrupts. Der Wert des Statusregisters wird vom Stack geholt. Anschließend wird noch der Inhalt des Programmzählers vom Stack geholt und ab dieser Adresse im Programm weitergemacht. Das unterbrochene Programm 'merkt' von alledem nichts, da sich nichts geändert hat. Der Prozessor selbst rettet dabei nur den Status - die anderen Register, falls sie in der Interruptroutine benutzt werden, müssen vom Programmierer selbst in Sicherheit gebracht werden und vor dem Rücksprung mit RTI wieder zurückgeholt werden, z.B. so

```
INTERRUPT       PHA  ; Akku retten
                TXA
                PHA  ; X-Register retten
                TYA
                PHA  ; Y-Register retten

                ...  ; Interrupt-Routine

                PLA
                TAY  ; Y-Register zurückholen
                PLA
                TAX  ; X-Register zurückholen
                PLA  ; Akku zurückholen
                RTI  ; Rückkehr ins unterbrochene Programm
```

Die Struktur einer Interrupt-Routine ist also der eines Unterprogramms ähnlich. Der Hauptunterschied ist jedoch der, daß ein Unterprogramm vom Hauptprogramm aus an einer bestimmten Stelle aufgerufen wird, während die Interruptroutine von außen per Hardware ausgelöst wird und an jeder beliebigen Stelle das Hauptprogramm unterbrechen kann. Im Gegensatz zu einem Unterprogrammaufruf wird bei einem Interrupt zusätzlich zur Rücksprungadresse noch der augenblickliche Prozessorstatus mit abgespeichert, da sonst das unterbrochene Programm anschließend nicht ordnungsgemäß weiterarbeiten könnte. Doch jetzt zu der wichtigsten Frage.

Durch was kann ein Interrupt ausgelöst werden?

Dazu gibt es in Ihrem Commodore 64 mehrere Möglichkeiten. Schauen wir uns dazu an, wodurch ein IRQ ausgelöst werden kann. Das können der

Videocontroller VIC 6569

und der I/O-Baustein

CIA 6526

Gemeint ist hier die CIA mit der Adresse $DC00.

Einen nicht maskierbaren Interrupt (NMI) auslösen können die

CIA 6526 (Adresse $DD00)

sowie die

RESTORE-Taste

Um erfolgreich eigene Interruptroutinen programmieren zu können, ist es unerläßlich, über die Möglichkeiten und Eigenschaften dieser Peripheriebausteine genau Bescheid zu wissen. Wir werden die verschiedenen Bausteine in soweit besprechen, wie es für unsere Programmierung erforderlich ist. Näheres können Sie dem Buch '64 intern' entnehmen.

2.2 Die CIA 6526

Die CIA (Complex Interface Adapter) 6526 ist ein Peripheriebaustein der 65XX-Familie, der über zwei 8-Bit-Ein/Ausgabeports, ein serielles 8-Bit-Schiebe-Register, zwei kaskadierbare 16-Bit-Timer, eine Echtzeituhr und diverse Steuerleitungen verfügt.

Die CIA hat 16 Register, die vom Prozessor wie fortlaufende Speicherzellen angesprochen werden können. Der Commodore hat gleich zwei dieser Bausteine; der erste ist unter den Adressen $DC00 bis $DC0F zu erreichen, der zweite läßt sich von $DD00 bis $DD0F ansprechen.

Auf den nächsten Seiten finden Sie nun eine kurze Beschreibung dieser 16 Kontrollregister, auf die wir bei den Programmen dann näher eingehen.

Register 0 Port Register A
Zugriff: READ/WRITE
Der Inhalt dieses Registers spiegelt den Zustand des Ein/Ausgabe-Ports A wieder.

Register 1 Port Register B
Zugriff: READ/WRITE
Der Inhalt dieses Registers spiegelt den Zustand des Ein/Ausgabe-Ports B wieder.

Register 2 Datenrichtungsregister A
Zugriff: READ/WRITE
Mit diesem Register kann jede der acht Leitungen des Port A auf Eingabe oder Ausgabe geschaltet werden. Dazu muß das entsprechende Bit des Datenrichtungsregisters 0 (=Eingabe) oder 1 (=Ausgabe) sein.

Register 3 Datenrichtungsregister B
Zugriff: READ/WRITE
Dieses Register hat dieselbe Funktion wie Register 2, allerdings für den Port B.

Register 4 Timer A Lo-Byte
Zugriff: READ
Beim Lesezugriff gibt der Inhalt dieses Registers den augenblicklichen Stand des Timers A (Lo-Byte) wieder.
Zugriff: WRITE
Mittels eines Schreibbefehls kann man in dieses Register das Lo-Byte des Wertes laden, von dem der Timer aus auf Null zählen soll.

Register 5 Timer A Hi-Byte

Zugriff: READ

Beim Lesezugriff gibt der Inhalt dieses Registers den augenblicklichen Stand des Timers A (Hi-Byte) wieder.

Zugriff: WRITE

Mittels eines Schreibbefehls kann man in dieses Register das Hi-Byte des Wertes laden, von dem der Timer aus auf Null zählen soll.

Register 6 Timer B Lo-Byte

Dieses Register entspricht in der Funktion dem Register 5, bezieht sich jedoch auf den Timer B.

Register 7 Timer B Hi-Byte

Dieses Register entspricht in der Funktion dem Register 6, bezieht sich jedoch auf den Timer B.

Register 8 Time of Day (Echtzeituhr) Zehntel Sekunden

Zugriff: READ

Beim Lesezugriff geben die Bits 0 bis 3 den augenblicklichen Stand der Echtzeituhr wieder, und zwar die Zehntelsekunden im BCD-Format. Die Bits 4-7 sind immer null.

Zugriff: WRITE

Beim Schreiben in das Register 8 können Sie abhängig von der Vorwahl durch das Kontroll-Register B (Register 15) entweder die Zehntelsekunden der Uhrzeit setzen (Uhr stellen) oder die Alarmzeit vorwählen. Die Zehntelsekunden müssen dabei im BCD-Format angegeben werden, die Bits 4 bis 7 müssen null sein.

Register 9 Time of Day Sekunden

Zugriff: READ

Beim Lesezugriff auf dieses Registers erhalten Sie die Sekunden der aktuellen Uhrzeit im BCD-Format. Die Bits 0 bis 3 stellen dabei die Einerstelle dar, während die Bit 4 bis 7 die Zehnerstelle repräsentieren.

Zugriff: WRITE

Beim Schreibzugriff können Sie analog dem Register 8 entweder die Uhrzeit setzen oder die Alarmzeit vorwählen. Die Sekundenzahl muß dabei im BCD-Format vorliegen.

Register 10 Time of Day Minuten

Das Register 10 ist analog zu Register 9 organisiert, ist jedoch für die Minuten zuständig.

Register 11 Time of Day Stunden

Zugriff: READ

Beim Lesen gibt diesen Register den aktuellen Stundenwert der Echtzeituhr wieder. Dabei repräsentieren die Bits 0 bis 3 die Einerstelle. Da die Uhr nur von eins bis zwölf Stunden zählt, ist für die Zehnerstelle nur ein Bit erforderlich, nämlich Bit 4. Bit 7 dient entsprechend der amerikanischen Zeitdarstellung als Flag für vormittags (AM, Bit 7=0) oder nachmittags (PM, Bit 7=1).

Zugriff: WRITE

Der Schreibzugriff geschieht analog den anderen Registern der Echtzeit, jedoch ist die Bedeutung der einzelnen Bits analog dem Lesezugriff.

Register 12 Serielles Schieberegister

In dieses Register werden die Daten geschrieben, die über den seriellen Port bitweise herausgeschoben werden. Beim Lesen können die hineingeschobenen Daten dort abgeholt werden.

Register 13 Interrupt Control Register

Zugriff: READ (Interrupt-Daten)

Bit 0 Unterlauf von Timer A

Bit 1 Unterlauf von Timer B

Bit 2 Gleichheit von Uhrzeit und gewählter Alarmzeit

Bit 3 Schieberegister voll (bei Eingabe) oder leer (bei Ausgabe)

Bit 4 Impuls am Pin FLAG aufgetreten

Bit 5-6 immer 0

Bit 7 Bit sieben ist dann gesetzt, wenn mindestens ein Bit der Bits 0 bis 4 sowohl im Interrupt Control Register als auch im Interrupt Masken Register auf eins ist.

Achtung! Beim Lesen dieses Registers werden alle Bits gelöscht!

Zugriff: WRITE (Interrupt-Maske)

Die Bedeutung der Bits 0 bis 4 ist gleich. Wenn zusätzlich Bit sieben gesetzt ist, so kann man damit den Interrupt für die gewählte Funktion freigeben.

Ist Bit sieben gelöscht, so löscht ein Eins-Bit die entsprechende Interruptmöglichkeit.

Register 14 Control Register A

Zugriff: READ/WRITE

Bit 0 0= Timer A Stop, =1 Timer A Start

Bit 1 1= Unterlauf von Timer A wird an PB6 signalisiert

Bit 2 0= jeder Unterlauf von Timer A erzeugt an PB6 einen Hi-Impuls, 1= jeder Unterlauf von Timer A invertiert den Zustand von PB6

Bit 3 1= Timer A zählt nur einmal vom Ausgangswert auf null und bleibt dann stehen (one shot), 0= Timer A startet nach jedem Unterlauf automatisch wieder (continuous mode)

Bit 4 1= unbedingtes Laden eines neuen Wertes von Timer A

Bit 5 0= Timer zählt Systemtaktpulse, 1= Timer zählt steigende Flanken an CNT

Bit 6 0= serieller Port ist Eingang, 1= serieller Port ist Ausgang

Bit 7 0= Echtzeituhr läuft mit 60 Hz, 1= Echtzeituhr läuft mit 50 Hz

Register 15 Control Register B

Zugriff: READ/WRITE

Bit 0 bis 4 Gleiche Bedeutung wie die entsprechenden Bits des Control Registers A, jedoch für Timer B und PB7

Bit 5 und 6 Diese Bits bestimmen die Triggerquelle von Timer B. 00= Timer zählt Systemtakte, 01= Timer B zählt steigende CNT-Flanken, 10= Timer B zählt Unterläufe von Timer A, 11= Timer B zählt Unterläufe von Timer A wenn CNT=1 ist.

Bit 7 0= Uhrzeit setzen, 1= Alarmzeit setzen.

2.3 Die Benutzung des Systeminterrupts

Die einfachste Möglichkeit, eine eigene Interruptroutine zu programmieren, besteht darin, sich einfach in den Systeminterrupt mit 'einzuhängen'. Wer löst den Systeminterrupt aus und welche Aufgaben hat er zu erfüllen?

Der Systeminterrupt ist von einem Timer in der CIA 1 gesteuert. Ein Timer ist einfach ein Zähler, der bei jedem Maschinentakt, das ist etwa jede Mikrosekunde, um eins vermindert wird. Ist der Timer bis auf Null heruntergezählt, so gibt er einen Impuls an den IRQ-Eingang des Prozessors. Das laufende Programm wird unterbrochen und es wird in eine Interruptroutine verzweigt, die ab Adresse $EA31 zu finden ist. Der Timer besteht aus zwei 8-Bit-Registern und kann deshalb Zeiten bis zu 2^{16} Mikrosekunden gleich 65 Millisekunden liefern. Der Systeminterrupt wird kontinuierlich alle sechzigstel Sekunden, das ist ca. alle 16 ms ausgelöst. Welche Aufgaben hat diese Routine zu erfüllen?

Als erstes wird geprüft, ob die STOP-Taste gedrückt ist. Ist dies der Fall, so wird ein Flag in der Zeropage gesetzt. Dieses Flag wird vor der Ausführung jedes BASIC-Statement geprüft. Ist es gesetzt, so wird das laufende BASIC-Programm abgebrochen. Die Routine zur Abfrage der STOP-Taste erhöht ebenfalls die interne Uhr TI, die die Zeit in sechzigstel Sekunden mißt. Die zweite Aufgabe betrifft den Cursor. Befindet sich der Rechner im Direktmodus oder wird eine Eingabe erwartet, so blinkt der Cursor. Dazu wird bei jedem 20. Aufruf der Interruptroutine die Darstellung des Zeichens, an dem der Cursor steht, invertiert. Der Cursor blinkt also 60/20 = 3 mal in der Sekunde. Eine weitere Aufgabe besteht in der Überwachung der Datasette. Falls die Datasette nicht unter Programmkontrolle ist (z.B. LOAD oder SAVE), so wird

abhängig davon, ob eine Taste an der Datasette gedrückt ist oder nicht, der Motor der Datasette ein- oder ausgeschaltet. Die letzte und vielleicht wichtigste Aufgabe der Interruptroutine besteht in der Abfrage der Tastatur. Ist eine Taste gedrückt, so wird der Tastenkode ermittelt und der Wert im Tastaturpuffer abgelegt. Dieser Tastaturpuffer ist zehn Zeichen lang. Dadurch ist es möglich, daß Sie schon mehrere Tasten 'im voraus' drücken können, die erst dann auf dem Bildschirm erscheinen, wenn das Programm die Zeichen erwartet. Die Anzahl der Zeichen im Tastaturpuffer wird ebenfalls vermerkt. Sind diese Aufgaben alle erledigt, wird die Interruptroutine verlassen und im unterbrochenen Programm weitergemacht.

Wie wir bereits erwähnten, holt sich der Prozessor die Adresse der Interruptroutine von den Speicherstellen $FFFE und $FFFF, also aus dem ROM. Wie können wir diese Werte ändern? Sehen wir uns dazu an, was nach dem Interrupt passiert. Die Adresse, auf die der Interruptvektor zeigt, ist $FF48.

```
******************************   IRQ-Einsprung
FF48  48        PHA
FF49  8A        TXA
FF4A  48        PHA              Register retten
FF4B  98        TYA
FF4C  48        PHA
FF4D  BA        TSX
FF4E  BD 04 01  LDA $0104,X      Break-Flag vom Stapel holen
FF51  29 10     AND #$10         und testen
FF53  F0 03     BEQ $FF58        nicht gesetzt ?
FF55  6C 16 03  JMP ($0316)      BREAK - Routine
FF58  6C 14 03  JMP ($0314)      Interrupt - Routine
```

Zuerst werden also die Inhalte der Register auf den Stack gerettet. Dann wird der Inhalt des Statusregisters, das beim Interrupt automatisch auf den Stack gerettet wurde, gelesen und Bit 4 isoliert. Dies ist das BREAK-Flag, das beim BRK-Befehl gesetzt wird. Der BRK-Befehl simuliert per Software einen Interruptaufruf. Um ihn von einem echten Interrupt zu unterscheiden, wird das BREAK-Flag gesetzt. Anhängig davon wird zu zwei indirekten Sprüngen verzweigt. War das Flag gesetzt, so wird über den Vektor $316/$317 gesprungen, bei einem echten Interrupt über den Vektor $314/$315.

Der Vektor $314/$315 ist der eigentliche Interrupt-Vektor und zeigt im Normalfall auf die oben erwähnte Adresse $EA31.

Möchten wir, daß innerhalb der Interruptroutine noch eine weitere Aufgabe erfüllt wird, so können wir folgendermaßen vorgehen:

Wir ändern den Interruptvektor so, daß er auf unsere eigene Routine zeigt. Ist unsere Routine abgearbeitet, so springen wir zur Systeminterruptroutine, damit diese Aufgaben weiterhin erfüllt werden. Mit diesem Verfahren können wir also einen zweiten 'Job' unabhängig vom Hauptprogramm sechzigmal in der Sekunde ausführen lassen. Diese Routine darf natürlich selbst nicht länger als eine sechzigstel Sekunde dauern, da sonst ja keine Zeit für das Hauptprogramm mehr übrig bliebe. Eine lange Interruptroutine macht sich in einer Verlangsamung des Hauptprogramms bemerkbar.

Was könnte der Rechner zusätzlich sechzigmal pro Sekunde ausführen? Hier sind Ihrer Phantasie keine Grenzen gesetzt. Sie könnten z.B. den Bildschirm oder eine Schrift auf dem Bildschirm blinken lassen, ähnlich wie dies mit dem Cursor

geschieht. Damit das Blinken nicht zu schnell geht, müssen wir wie beim Cursorblinken einen Zähler benutzen, der es uns gestattet, das Umschalten der Farbe nur bei z.B. jedem 30. Aufruf durchzuführen.

```
PROFI-ASS 64 V2.0    SEITE 1

100:    C000                      .OPT P,00
110:                    ;
120:                    ; HINTERGRUND/RAHMEN BLINKEN
130:                    ;
140:    C000                      *=   $C000
150:                    ;
160:    D020            BORDER    =    $D020    ; RAHMEN
170:    D021            BACK      =    $D021    ; HINTERGRUND
180:    EA31            IRQROUT   =    $EA31
190:    0314            IRQVEC    =    $314
200:                    ;
210:    001E            ANZAHL    =    30       ; ALLE HALBE SEKUNDE
220:                    ;
230:    C000 78         INIT      SEI           ; INTERRUPT VERHINDERN
240:    C001 A9 0D                LDA  #<BLINK
250:    C003 A0 C0                LDY  #>BLINK
260:    C005 8D 14 03             STA  IRQVEC   ; IRQ-VECTOR AUF BLINK-ROUTINE
270:    C008 8C 15 03             STY  IRQVEC+1
280:    C00B 58                   CLI
290:    C00C 60                   RTS
300:                    ;
310:    C00D CE 26 C0 BLINK       DEC  COUNT    ; ZAEHLER VERMINDERN
320:    C010 D0 11                BNE  FERTIG
330:    C012 A9 1E                LDA  #ANZAHL
340:    C014 8D 26 C0             STA  COUNT    ; ZAEHLER NEU SETZEN
350:                    ; FARBEN VERTAUSCHEN
360:    C017 AE 21 D0             LDX  BACK
```

```
PROFI-ASS 64 V2.0    SEITE 2

370:    C01A AD 20 D0           LDA  BORDER
380:    C01D 8D 21 D0           STA  BACK
390:    C020 8E 20 D0           STX  BORDER
400:                   ;
410:    C023 4C 31 EA FERTIG    JMP  IRQROUT
420:                   ;
430:    C026 1E        COUNT    .BYT ANZAHL  ; ZAEHLER
]C000-C027
NO ERRORS
```

Sehen wir uns das Programm einmal genauer an. Die Routine INIT besorgt die Initialisierung und setzt den Interruptvektor auf unsere Blinkroutine. Bitte beachten Sie, daß während der Änderung des Interruptvektors ein möglicher Interrupt mit dem Befehl SEI gesperrt wird. Würde nämlich ein Interrupt ausgelöst, wenn das Lo-Byte schon auf den neuen Wert zeigt, während das Hi-Byte des IRQ-Vektors noch auf die alte Routine weist, so würde an eine undefinierte Stelle gesprungen und der Rechner wird 'abstürzen'. Ist auch das H-Byte gesetzt, so wird der Interrupt mit CLI wieder freigegeben und wir kehren mit RTS zurück. Ab jetzt ist die neue Interrupt-Routine aktiv.

Beim nächsten Interruptaufruf passiert folgendes. Zuerst wird die Speicherstelle COUNT um eins vermindert. Hat sich kein Wert von Null ergeben, so wird zum Label FERTIG verzweigt und von da aus wird die normale Interruptroutine ausgeführt. War der Zähler jedoch auf Null, so wird er wieder mit dem Wert 30 geladen. Anschließend werden die Farben des Rahmens und des Hintergrunds vertauscht, was zu dem Blinkeffekt führt.

Aktivieren können wir unsere Routine durch den Aufruf von SYS 12*4096. Ab sofort blinkt der Bildschirm zweimal in der Sekunde. Dieses Interruptprogramm läuft nun völlig unabhängig von einem BASIC- oder Maschinenprogramm solange, bis der Interruptvektor wieder auf den alten Wert zurückgesetzt wird. Dies kann z.B. durch Drücken von RUN/STOP-RESTORE geschehen.

Die Blinkfrequenz können wir einfach mit dem Label ANZAHL ändern; es gibt an, nach wieviel sechzigstel Sekunden jeweils die Farben vertauscht werden sollen.

Als zweites Beispiel einer Interruptroutine wollen wir den Cursor einmal anders gestalten. Der Cursor soll nicht blinken, sondern nur durch ein inverses Zeichen dargestellt werden. Dazu können wir nicht einfach unsere neue Routine vor die normale Interruptroutine setzen, sondern wir müssen den Teil ersetzen, der für das Cursorblinken zuständig ist.

```
******************************   Interrupt-Routine
EA31   20 EA FF   JSR $FFEA      Stop-Taste, Zeit erhöhen
EA34   A5 CC      LDA $CC        Blink-Flag für Cursor
EA36   D0 29      BNE $EA61      nicht blinkend, dann weiter
EA38   C6 CD      DEC $CD        Blinkzähler erniedrigen
EA3A   D0 25      BNE $EA61      nicht null, dann weiter
EA3C   A9 14      LDA #$14       Blinkzähler wieder auf 20 setzen
EA3E   85 CD      STA $CD        und merken
EA40   A4 D3      LDY $D3        Cursorspalte
EA42   46 CF      LSR $CF        Blinkschalter null dann C=1
EA44   AE 87 02   LDX $0287      Farbe unter Cursor
EA47   B1 01      LDA ($D1),Y    Zeichen-Kode setzen
EA49   B0 11      BCS $EA5C      Blinkschalter war ein, dann weiter
EA4B   E6 CF      INC $CF        Blinkschalter ein
EA4D   85 CE      STA $CE        Zeichen unter Cursor merken
EA4F   20 24 EA   JSR $EA24      Zeiger in Farb-RAM berechnen
EA52   B1 F3      LDA ($F3),Y    Farb-Kode holen
```

```
EA54    8D 87 02    STA $0287     und merken
EA57    AE 86 02    LDX $0286     Farb-Kode unter Cursor
EA5A    A5 CE       LDA $CE       Zeichen unter Cursor
EA5C    49 80       EOR #$80      RVS-Bit umdrehen
EA5E    20 1C EA    JSR $EA1C     Cursorzeichen und -Farbe setzen
```

Das Cursorblinken wird also folgendermaßen realisiert. Zuerst wird geprüft, ob der Cursor überhaupt aktiv ist. Falls nicht, so wird der folgende Teil übersprungen. Dann wird der Blinkzähler dekrementiert. Ist er nicht auf null, so wird ebenfalls der folgende Teil übersprungen. Ansonsten wird geprüft, ob der Cursor gerade in der invertierten Phase war. Abhängig davon wird der augenblickliche oder gespeicherte Wert invertiert und dargestellt. Im Farbram geschieht das gleiche mit der Zeichenfarbe und der augenblicklichen Cursorfarbe.

Wir wollen die Routine nun so modifizieren, daß wir einen stehenden Cursor haben. Dies können wir mit dem folgenden Programm bewerkstelligen.

```
PROFI-ASS 64 V2.0      SEITE 1

100:   C000                     .OPT P,00
110:                  ;
120:                  ; CURSOR MODIFIZIEREN
130:                  ;
140:   FFEA           STOP     =    $FFEA    ; Stoptaste abfragen
150:   00CC           CURSFLAG =    $CC      ; Flag für sichtbaren Cursor
160:   00CF           INVERS   =    $CF      ; Flag für invertiertes Zeichen
170:   0287           CURSCOL  =    $287     ; Farbe unter Cursor
180:   00CE           CURSCHAR =    $CE      ; Zeichen unter Cursor
190:   00D1           CHAR     =    $D1      ; Zeiger in Videoram
200:   00F3           COLOR    =    $F3      ; Zeiger in Farbram
```

```
210:   EA24          SETCOL    =    $EA24    ; Zeiger auf Farbram setzen
220:   00D3          SPALTE    =    $D3      ; Cursorspalte
230:   0286          COLSTR    =    $286     ; Cursorfarbe
240:   0314          IRQVEC    =    $314     ; IRQ-Vektor
250:   EA61          CONTIRQ   =    $EA61    ; IRQ fortführen
260:                 ;
270:   C000 78       INIT      SEI           ; Interrupt verhindern
280:   C001 A9 0D              LDA  #<NEWCURS
290:   C003 A0 C0              LDY  #>NEWCURS
300:   C005 8D 14 03           STA  IRQVEC   ; IRQ-Vektor auf neue Routine
310:   C008 8C 15 03           STY  IRQVEC+1
320:   C00B 58                 CLI
330:   C00C 60                 RTS
340:                 ;
350:   C00D 20 EA FF NEWCURS   JSR  STOP       ; Stoptaste testen
360:   C010 A5 CC              LDA  CURSFLAG   ; Cursor sichtbar ?
370:   C012 D0 1D              BNE  NOCURSOR   ; nein
380:   C014 A4 D3              LDY  SPALTE     ; Cursorspalte
390:   C016 A5 CF              LDA  INVERS     ; Zeichen schon invertiert ?
400:   C018 D0 17              BNE  NOCURSOR   ; ja
410:   C01A E6 CF              INC  INVERS     ; Flag für invers setzen
420:   C01C 20 24 EA           JSR  SETCOL     ; Zeiger in Farbram
430:   C01F B1 D1              LDA  (CHAR),Y   ; Zeichen an Cursorposition
430:   C021 85 CE              STA  CURSCHAR   ; merken
440:   C023 49 80              EOR  #$80       ; RVS-Bit flippen
450:   C025 91 D1              STA  (CHAR),Y   ; und in Videoram
460:   C027 B1 F3              LDA  (COLOR),Y  ; Farbe
460:   C029 8D 87 02           STA  CURSCOL    ; merken
470:   C02C AD 86 02           LDA  COLSTR     ; Cursorfarbe
480:   C02F 91 F3              STA  (COLOR),Y  ; setzen
490:   C031 4C 61 EA NOCURSOR  JMP  CONTIRQ    ; IRQ fortsetzen
]C000-C034
NO ERRORS
```

Wenn Sie diese Routine mit SYS 12*4096 aktivieren, so wird der Cursor lediglich durch ein invertiertes Zeichen dargestellt. Sie können diese Routine jedoch nach Ihrem eigenen Geschmack modifizieren; z.B. braucht für das Zeichen nicht die aktuelle Cursorfarbe genommen werden, sondern z.B. immer die Farbe weiß. Auch anstelle der Inversdarstellung durch Invertieren des obersten Bits könnten Sie etwas anders machen, z.B. ein Unterstreichungszeichen darstellen. Ebenso wäre es möglich, das Zeichen unverändert zu lassen und lediglich zwischen zwei verschiedenen Farben umzuschalten. Betrachten Sie diese Beispiele bitte nur als Anregung für Ihre eigenen Experimente mit der Interruptroutine.

Hier können wir auch noch kurz auf eine Möglichkeit eingehen, die STOP-Taste zu blockieren, die Sie vielleicht schon kennen. Da die Abfrage auf die STOP-Taste als erstes in der Interruproutine geschieht, können wir durch Setzen des Interruptvektors auf die Adresse dahinter diesen Test umgehen und ein laufendes BASIC-Programm kann nicht mehr mit der STOP-Taste unterbrochen werden:

```
POKE 788, PEEK(788)+3
```

Der Vektor wird einfach um drei Bytes erhöht, so daß der Test übersprungen wird. Ein Nachteil bei dieser Methode ist jedoch, daß die interne Uhr TI und TI$ dadurch stehenbleibt. Das Weiterzählen der Uhr um eine sechzigstel Sekunde geschieht nämlich ebenfalls in dieser Routine.

Eine weitere Anwendung des Systeminterrupts ist die Auslösung einer bestimmten Aktion lediglich auf einen Tastendruck hin. Z.B. ist es möglich, durch Drücken einer Funktionstaste eine Hardcopyroutine aufzurufen, die den Bildschirminhalt auf einem Drucker ausgibt.

Dazu wird in der Interruptroutine geprüft, ob die Taste gedrückt ist. Ist das der Fall, so kann eine Routine aufgerufen werden, die die spezielle Aufgabe ausführt. Auch hier sind wieder manigfaltige Anwendungen möglich, z.B. könnte zwischen zwei Bildschirmseiten umgeschaltet werden. Versuchen wir, diese Möglichkeit einmal zu realisieren.

```
PROFI-ASS 64 V2.0     SEITE 1

100:   033C                      .OPT P1,00
110:                 ;
120:                 ; BILDSCHIRMSEITEN UMSCHALTEN
130:                 ;
140:   0003          PNT1      =   3
150:   0005          PNT2      =   5
160:   DD00          VIDEOMAP  =   $DD00    ; 16K Video-Bereich
170:   0288          VIDEOPGE  =   648
180:   0314          IRQVEC    =   $314
190:   EA31          IRQALT    =   $EA31
200:   D000          CHARGEN   =   $D000    ; Charactergenerator
210:   D800          COLOR     =   $D800    ; Farbram
220:   C000          COLOR2    =   $C000    ; Speicher für Farbram
230:   0001          PORT      =   1        ; Prozessorport
240:   028D          CTRL      =   653      ; Flag für CONTROL
250:   00C5          KEY       =   $C5      ; letzte Taste
260:   0004          F1        =   4        ; Matrixnummer der F1-Taste
270:                 ;
280:   033C                *=  828
```

```
290:                     ;
300:    033C 78          INIT     SEI
310:    033D 20 94 03             JSR  SETCHAR  ; Charactergenerator kopieren
320:    0340 A9 4C                LDA  #< TEST
330:    0342 A0 03                LDY  #> TEST
340:    0344 8D 14 03             STA  IRQVEC   ; Zeiger auf neue Routine
350:    0347 8C 15 03             STY  IRQVEC+1
360:    034A 58                   CLI
370:    034B 60                   RTS
380:                     ;
390:    034C AD 8D 02 TEST        LDA  CTRL     ; Controltaste gedrückt ?
400:    034F 29 04                AND  #%100
410:    0351 F0 09                BEQ  NOSWITCH ; nein
420:    0353 A5 C5                LDA  KEY      ; F1 gedrückt ?
430:    0355 C9 04                CMP  #F1
440:    0357 D0 03                BNE  NOSWITCH ; nein
450:    0359 20 5F 03             JSR  SWITCH   ; Seiten vertauschen
460:    035C 4C 31 EA NOSWITCH JMP  IRQALT
470:                     ;
480:    035F A0 00       SWITCH   LDY  #0
490:    0361 84 03                STY  PNT1
490:    0363 84 05                STY  PNT2
500:    0365 A9 D8                LDA  #>COLOR  ; Zeiger auf Farbram
500:    0367 85 04                STA  PNT1+1
510:    0369 A9 C0                LDA  #>COLOR2 ; Zeiger auf Speicher für Farbe
510:    036B 85 06                STA  PNT2+1
520:    0360 A2 04                LDX  #4      ; Anzahl Pages
530:    036F B1 03       SWAP     LDA  (PNT1),Y
540:    0371 48                   PHA
550:    0372 B1 05                LDA  (PNT2),Y
560:    0374 91 03                STA  (PNT1),Y  ; Farbspeicher austauschen
570:    0376 68                   PLA
```

```
580:   0377 91 05              STA  (PNT2),Y
590:   0379 C8                 INY
600:   037A D0 F3              BNE  SWAP
610:   037C E6 04              INC  PNT1+1
610:   037E E6 06              INC  PNT2+1
620:   0380 CA                 DEX             ; nächste Page
630:   0381 D0 EC              BNE  SWAP
640:   0383 AD 00 DD           LDA  VIDEOMAP
650:   0386 49 03              EOR  #%11     ; Zugriffsadresse für VIC
660:   0388 80 00 DD           STA  VIDEOMAP
670:   038B AD 88 02           LDA  VIDEOPGE
680:   038E 49 C0              EOR  #$C0     ; Bildschirmpage
690:   0390 8D 88 02           STA  VIDEOPGE
700:   0393 60                 RTS
710:                  ;
720:   0394 A0 00     SETCHAR  LDY  #0
730:   0396 84 03              STY  PNT1
740:   0398 A9 D0              LDA  #> CHARGEN
750:   039A 85 04              STA  PNT1+1
760:   039C A2 10              LDX  #$10
770:   039E A9 33     LOOP     LDA  #$33
780:   03A0 85 01              STA  PORT     ; Char-Generator einschalten
790:   03A2 B1 03              LDA  (PNT1),Y
800:   03A4 48                 PHA
810:   03A5 A9 30              LDA  #$30
820:   03A7 85 01              STA  PORT     ; RAM einschalten
830:   03A9 68                 PLA
840:   03AA 91 03              STA  (PNT1),Y
850:   03AC C8                 INY
860:   03AD D0 EF              BNE  LOOP
870:   03AF E6 04              INC  PNT1+1   ; nächste Page
880:   03B1 CA                 DEX
```

```
890:    03B2 D0 EA          BNE  LOOP
900:    03B4 A9 37          LDA  #$37    ; Standard Konfiguration
910:    03B6 85 01          STA  PORT
920:    03B8 60             RTS
]033C-03B9
NO ERRORS
```

Wir schalten dabei zwischen zwei Bildschirmseiten um. Die erste Seite liegt wie gewöhnlich bei $400, während wir die zweite $eite nach Adresse $C400 gelegt haben. Es ist dabei auch möglich, für die zweite $eite eigene Sprites zu aktivieren. Die Spritepointer müssen dann ab Adresse $C7F8 stehen und beziehen sich alle auf die Basisadresse $C000. Für Sprites steht Ihnen z.B. der Adressraum von $C800 bis $CFFF zur Verfügung, der Platz für 32 verschiedene Spritemuster bietet (Spritenummer 32 bis 63). Da das Farbram vom Videocontroller immer an der gleichen Adresse $D800 erwartet wird, speichern wir die Farbe der nicht angezeigten Seite von $C000 bis $C3FF. Des weiteren müssen wir beachten, daß der VIC innerhalb der obersten 16 K von $C000 bis $FFFF nicht auf das Charactergenerator-ROM zugreifen kann. Wir kopieren daher bei der Initialisierung den Charactergenerator vom ROM in das an der gleichen Adresse liegende RAM.

Die eigentliche Interruptroutine prüft das Bit 2 im Flag für die Controltaste. Ist dieses Bit gesetzt, so war die Controltaste gedrückt. Wenn dann zusätzlich noch die gedrückte F1-Taste erkannt wird, so wird die Routine abgerufen, die die Farbspeicher vertauscht sowie die Parameter zur Anzeige der jeweils anderen Bildschirmseite setzt. Anschließend wird in die normale Interruptroutine verzweigt.

Wenn Sie unser Programm assembliert und mit SYS 828 aktiviert haben, so können Sie durch gleichzeitiges Drücken von CTRL- und F1-Taste auf eine zweite Bildschirmseite umschalten. Beim ersten Mal sollten Sie den Bildschirm löschen, da noch zufällige Werte im Videoram stehen. Durch nochmaliges Drücken der beiden Tasten schalten Sie wieder auf die ursprüngliche Seite zurück. Der Cursor bleibt dabei an der gleichen Stelle stehen.

Als weitere Anregung könnten Sie einmal versuchen, während der Interruptroutine die Uhrzeit anzuzeigen. Sie haben dadurch ständig die aktuelle Uhrzeit unabhängig von anderen Programmaktivitäten auf dem Bildschirm. Eine derartige Routine finden Sie in dem Buch '64 Tips & Tricks'.

Eine sicherlich ebenso interessante Möglichkeit einer Interruptroutine gibt es in Zusammenhang mit den Sprites. Bei jedem Interrupt könnten ein oder mehrere Sprites bewegt werden. Analog bietet sich die Interruptroutine auch im Zusammenhang mit der Programmierung des Sound-Chips an. Hier können Sie Soundsequenzen oder komplette Musikstücke unabhängig vom anderen Programmablauf ablaufen lassen. Sie sehen, die Möglichkeiten, die sich Ihnen hier bieten, sind nahezu unerschöpflich. Ehe wir nun selbst Interrupts auslösen, noch zwei Routinen, die in den Systeminterrupt eingebunden sind.

Wenn Sie den Userport zum Anschluß eigener Geräte benutzen, kann das folgende Programm für Sie nützlich sein. Es ist in den Systeminterrupt eingebunden und zeigt Ihnen ständig den Zustand der einzelnen Bits des Userport auf dem Bildschirm an. In der ersten Bildschirmzeile wird das Richtungsregister dargestellt. Daraus können Sie entnehmen, welche Leitungen

auf Eingang (=0) oder auf Ausgang (=1) geschaltet sind. In der Zeile darunter wird der Zustand der Userportleitungen dargestellt; eine 0 bedeutet Lo-Pegel, ein Hi-Pegel wird durch eine 1 ausgedrückt. Beide Anzeigen finden Sie dahinter noch in der Hexdarstellung.

```
PROFI-ASS 64 V2.0    SEITE 1

033C                        .OPT P1,00
                ;
                ; ANZEIGE DES USER-PORTS
                ;
DD00            CIA2     =    $DD00
DD01            USERPORT =    CIA2+1
DD03            RICHTUNG =    CIA2+3
                ;
0288            VIDEOPGE =    648
D800            COLORRAM =    $D800
                ;
0007            FARBE    =    7        ; Gelb
                ;
0314            IRQVEC   =    $314
EA31            IRQALT   =    $EA31
00FB            PNT      =    $FB
                ;
033C                     *=   828
033C 78         INIT     SEI
033D AD 14 03            LDA  IRQVEC
0340 49 7E               EOR  #< IRQALT ^ ANZEIGE
0342 80 14 03            STA  IRQVEC
0345 AD 15 03            LDA  IRQVEC+1
0348 49 E9               EOR  #> IRQALT ^ ANZEIGE
034A 8D 15 03            STA  IRQVEC+1
034D 58                  CLI
```

```
360:  034E 60                RTS
370:                  ;
380:  034F A5 FB     ANZEIGE  LDA  PNT
390:  0351 48                 PHA            ; Zeiger retten
400:  0352 A5 FC              LDA  PNT+1
410:  0354 48                 PHA
420:  0355 A9 1C              LDA  #28
430:  0357 85 FB              STA  PNT       ; Zeiger in Videoram
440:  0359 AD 88 02           LDA  VIDEOPGE
450:  035C 85 FC              STA  PNT+1
460:  035E AD 03 DD           LDA  RICHTUNG
470:  0361 A0 00              LDY  #0        ; Richtung in oberster Zeile
480:  0363 20 77 03           JSR  DISPLAY ; anzeigen
490:  0366 AD 01 DD           LDA  USERPORT
500:  0369 A0 28              LDY  #40       ; Userport in zweiter Zeile
510:  036B 20 77 03           JSR  DISPLAY ; anzeigen
520:  036E 68                 PLA
530:  036F 85 FC              STA  PNT+1     ; Zeiger zurückholen
540:  0371 68                 PLA
550:  0372 85 FB              STA  PNT
560:  0374 4C 31 EA           JMP  IRQALT    ; zum normalen IRQ
570:                  ;
580:  0377 48        DISPLAY  PHA            ; Wert für Hex-Anzeige merken
590:  0378 A2 08              LDX  #8
600:  037A 0A        LOOP     ASL            ; oberstes Bit ins Carry
610:  037B 48                 PHA
620:  037C A9 30              LDA  #"0"      ; Null anzeigen
630:  037E 90 02              BCC  NULL
640:  0380 A9 31              LDA  #"1"      ; wenn C=1 dann Eins anzeigen
650:  0382 91 FB     NULL     STA  (PNT),Y
660:  0384 A9 07              LDA  #FARBE    ; und Farbe setzen
670:  0386 99 1C D8           STA  COLORRAM+28,Y
```

```
PROFI-ASS 64 V2.0    SEITE 3

680:   0389 C8               INY
690:   038A 68               PLA
700:   038B CA               DEX           ; nächstes Bit
710:   038C D0 EC            BNE  LOOP
720:                 ;
730:                 ; HEXANZEIGE
740:                 ;
750:   038E C8               INY
760:   038F 68               PLA
770:   0390 48               PHA
780:   0391 4A               LSR
780:   0392 4A               LSR           ; oberes Nibble nach unten schieben
780:   0393 4A               LSR
780:   0394 4A               LSR
790:   0395 20 99 03         JSR  ASCII    ; HI-NIBBLE
800:   0398 68               PLA           ; UND LO-NIBBLE
810:   0399 29 0F   ASCII    AND  #%1111
820:   039B C9 0A            CMP  #10
830:   0390 90 02            BCC  KLEINER
840:   039F 69 06            ADC  #6
850:   03A1 69 30   KLEINER  ADC  #"0"     ; nach ASCII wandeln
860:   03A3 29 3F            AND  #%111111  ; in Bildschirmkode wandlen
870:   03A5 91 FB            STA  (PNT),Y
880:   03A7 A9 07            LDA  #FARBE  ; und Farbe setzen
890:   03A9 99 1C D8         STA  COLORRAM+28,Y
900:   03AC C8               INY
910:   03AD 60               RTS
]033C-03AE
NO ERRORS
```

Die Initialisierung haben wir diesmal etwas anders gelöst. Wir verknüpfen den alten Wert des IRQ-Vektors exklusiv oder

mit dem neuen Wert und erreichen dadurch bei jedem Aufruf von SYS 828 eine Umschaltung des IRQ-Vektors zwischen dem alten Wert $EA31 und unserer neuen Routine ANZEIGE. Wenn Sie also die Anzeige abschalten wollen, geben Sie einfach nochmal SYS 828 ein und der Interruptvektor wird wieder auf $EA31 gesetzt

Das Programm selbst besteht aus einem Hauptprogramm, das zu Beginn die benötigten Speicherzellen auf den Stack rettet, so daß andere Programme, die diese Adressen ebenfalls benutzen, nicht beeinträchtigt werden. Dann wird der Zeiger PNT auf die 28. Spalte in der ersten Bildschirmzeile gesetzt, der Wert des Datenrichtungsregisters geladen und das Unterprogramm zur Anzeige aufgerufen. Danach wird Y auf 40 gesetzt, damit die Anzeigeroutine nun eine Zeile tiefer schreibt, und der Inhalt des Userports übergeben. Jetzt werden die Zeiger wieder zurückgeholt und es kann in die normale IRQ-Routine gesprungen werden.

Das Anzeigeprogramm soll den Wert im Akku einmal binär und zum zweiten hexdezimal darstellen. Zur Binärdarstellung benutzen wir eine Schleife über die 8 Bitpositionen. Bei jedem Schleifendurchlauf wird das jeweils oberste Bit mit ASL ins Carry geschoben. War dieses Bit eine '1', so ist das Carry gesetzt und wir geben eine '1' auf dem Bildchirm aus, ansonsten eine '0'. Nach der Binäranzeige wird der auf dem Stack zwischengespeicherte Wert nun hexadezimal angezeigt. Dazu wird das obere Nibble um vier Bits nach rechts in das untere Nibble geschoben, dann in die ASCII-Darstellung gewandelt und auf dem Bildschirm dargestellt. Das gleiche passiert dann noch mit dem unteren Nibble.

Wenn Sie unsere Routine mit SYS 828 aktivieren, erscheint z.B. folgende Darstellung auf dem Bildschirm:

```
00000000 00
11111111 FF
```

Dies ist der Wert nach dem Einschalten des Rechners. Der Userport ist auf Eingang geschaltet und die offenen Eingänge liefern einen Hi-Pegel. Schalten Sie den Userport auf Ausgabe und beschreiben Sie ihn mit 100.

```
POKE 56579, 255
POKE 56577, 100
```

Sie erhalten folgende Anzeige:

```
11111111 FF
01100100 64
```

Die Bits 2, 5 und 6 sind also gesetzt; dies entspricht der Hexzahl $64.

Die nächste Routine arbeitet ähnlich. Sie soll uns ständig über den zur Verfügung stehenden Speicherplatz auf dem laufenden halten. Wir vollziehen daher bei jedem Interrupt die FRE-Funktion nach. Dabei berechnen wir lediglich die Differenz zwischen dem Ende der Arrayvariablen und dem Beginn der Strings. Im Gegensatz zur echten FRE-Funktion wird in der Interruptroutine kein Garbage Collect durchgeführt. Dies ist einmal zu zeitaufwendig, zum anderen würde es auch die Situation verfälschen. Wollen wir den berechneten freien Speicherplatz dezimal anzeigen, so wäre eine Umwandlung ins Fließkommaformat und weiter in die ASCII-Darstellung erforderlich. Dies braucht zum einen Zeit, die wir jedoch noch verschmerzen könnten. Der Hauptnachteil einer solchen Methode besteht jedoch darin, daß wir sämtliche benutzte Speicherzellen dabei auf den Stack retten müßten, da der

Interrupt ja das laufende BASIC-Programm an jeder Stelle unterbrechen kann. Wir hätten ca. 20 oder mehr Speicherplätze zu retten, was einerseits viel Zeit und andererseits viel Platz im Stack erfordern würde, den wir möglicherweise gar nicht mehr zur Verfügung haben. Wir zeigen daher den freien Speicherplatz einfach in Hexadezimaldarstellung an. Dies ist genauso informativ und bedeutend schneller.

```
PROFI-ASS 64 V2.0     SEITE 1

100:   033C                      .OPT P,00
110:                  ;
120:                  ; ANZEIGE DES FREIEN SPEICHERPLATZES
130:                  ;
140:   0031           ARRAYEND =    $31
150:   0033           STRGSTRT =    $33
160:                  ;
170:   0400           VIDEO    =    1024
180:   D800           COLOR    =    $D800
190:                  ;
200:   0007           FARBE    =    7        ; Gelb
210:                  ;
220:   0314           IRQVEC   =    $314
230:   EA31           IRQALT   =    $EA31
240:                  ;
250:   033C           INIT     *=   828
260:   033C 78                 SEI
270:   0330 A9 49              LDA  #< FREE
280:   033F A0 03              LDY  #> FREE
290:   0341 80 14 03           STA  IRQVEC
300:   0344 8C 15 03           STY  IRQVEC+1
310:   0347 58                 CLI
320:   0348 60                 RTS
330:                  ;
```

```
340:   0349 38         FREE     SEC
350:   034A A5 33               LDA  STRGSTRT
360:   034C E5 31               SBC  ARRAYEND
370:   034E 08                  PHP
380:   034F A0 25               LDY  #37
390:   0351 20 61 03            JSR  ANZEIGE
400:   0354 28                  PLP
410:   0355 A5 34               LDA  STRGSTRT+1
420:   0357 E5 32               SBC  ARRAYEND+1
430:   0359 A0 23               LDY  #35
440:   035B 20 61 03            JSR  ANZEIGE
450:   035E 4C 31 EA            JMP  IRQALT
470:   0361 48         ANZEIGE  PHA
480:   0362 4A                  LSR
480:   0363 4A                  LSR
480:   0364 4A                  LSR
480:   0365 4A                  LSR
490:   0366 20 6A 03            JSR  ASCII
500:   0369 68                  PLA
510:   036A 29 0F      ASCII    AND  #%1111
520:   036C C9 0A               CMP  #10
530:   036E 90 02               BCC  KLEINER
540:   0370 69 06               ADC  #6
550:   0372 69 30      KLEINER  ADC  #"0"
560:   0374 29 3F               ANO  #%111111
570:   0376 99 00 04            STA  VIDEO,Y
580:   0379 A9 07               LDA  #FARBE
590:   037B 99 00 D8            STA  COLOR,Y
600:   037E C8                  INY
610:   037F 60                  RTS
]033C-0380
NO ERRORS
```

Nach Aufruf der Routine mit SYS 828 sind Sie ständig über den freien Speicherplatz im Bilde. Probieren Sie einmal folgendes BASIC-Programm.

```
100 DIM A$(200)
110 FOR I=1 TO 200 : A$(I) = CHR$(1) : NEXT
```

und starten es mit RUN. Sie können sehr schön sehen, wie der zur Verfügung stehenede Speicherplatz immer weniger wird. Geben Sie jetzt ?FRE(0) ein. Während der ca. 4 Sekunden, die die Funktion benötigt, können Sie sehr schön beobachten, wie sich der freie Speicherplatz ständig ändert.

Wenn Sie mit PROFI-ASS 64 arbeiten, können Sie ebenfalls sehen, wie in Pass 1 die Symboltabelle erstellt wird, weil dazu die gleichen Zeiger wie in BASIC benutzt werden.

2.4 Interrupts durch den Videocontroller

Nachdem wir den timergesteuerten Systeminterrupt zusätzlich für unsere eigenen Zwecke benutzt haben, wollen wir jetzt versuchen, selbst einen Interrupt auszulösen und entsprechende Routinen daraufhin auszuführen.

Dazu sehen wir uns nocheinmal die Chips an, die in der Lage sind, einen Interrupt auszulösen. Das sind zum einen die beiden CIAs 6526, wobei CIA 1 einen IRQ und CIA 2 einen NMI auslösen kann. Der Videocontroller VIC 6569 kann ebenfalls einen Interrupt auslösen. Die für den Interrupt zuständigen Register finden Sie in der folgenden Beschreibung.

Register 18 Zugriff READ

Beim Lesezugriff erhalten Sie die Nummer der Rasterzeile, die gerade auf dem Bildschirm dargestellt wird. Da die Nummer der Rasterzeile größer als 255 sein kann, wird Register 17 Bit 7 für den Übertrag benutzt.

Zugriff WRITE

Wenn Sie dieses Register beschreiben, so können Sie damit die Rasterzeile festlegen, bei deren Darstellung der VIC einen IRQ auslösen soll.

Register 25 Interrupt Request Register

Dieses Register signalisiert eine Interruptanforderung durch den VIC. Die einzelnen Bits stehen jeweils für eine Interruptquelle.

Bit 0 Der Videocontroller beschreibt die Rasterzeile, die in Register 18 geschrieben wurde.

Bit 1 Ein Sprite hat ein Hintergrundzeichen berührt. Die Nummer des Sprites wird in

Register 31 vermerkt.

Bit 2 Zwei Sprites sind kollidiert. Die Nummern der beteiligten Sprites werden in Register 30 vermerkt.

Bit 3 Am Lightpen wurde ein Strobe ausgelöst. Die X- und Y-Position wird dabei Register 19 und 20 vermerkt.

Bit 7 Dieses Bit wird zusammen mit einem der anderen Bits gesetzt.

Register 26 Interrupt Mask Register

Die Bedeutung der Bits entspricht dem Register 25. Ein Interrupt wird nur dann ausgelöst, wenn das entsprechende Bit im Interrupt Mask Register gesetzt ist und der Interrupt damit freigegeben ist.

Register 30 Sprite-Sprite-Kollision

Wenn zwei Sprites kollidieren, so werden die Bits gesetzt, die den Nummern der beteiligten Sprites entsprechen. Außerdem wird Bit 2 in Register 25 gesetzt. Nach dem Bearbeiten des Ereignisses müssen diese Bits wieder zurückgesetzt werden.

Register 31 Sprite-Hintergrund-Kollision

Stößt ein Sprite mit einem Hintergrundzeichen zusammen, so wird die Nummer des Sprites in diesem Register vermerkt und gleichzeitig Bit 1 in Register 25 gesetzt. Auch dieses Register muß anschließend zurückgesetzt werden.

Der Videocontroller kann also auf vier verschiedene Ereignisse hin einen Interrupt auslösen:

* Rasterzeile
* Sprite-Hintergrund-Kollosion
* Sprite-Sprite-Kollision
* Light-Pen

Der Videocontroller vermerkt in Register 25, ob eins der vier verschiedenen Ereignisse aufgetreten ist. Ob daraus jedoch eine Interruptanforderung an den Prozessor wird, entscheidet das Interrupt Mask Register. Erst wenn in diesem Register ein Bit gesetzt ist, wird bei einem entsprechenden Ereignis ein Interrupt ausgelöst. Dieses Register läßt sich jedoch nicht wie eine RAM-Speicherzelle lesen und beschreiben. Wenn Sie ein Bit setzen oder löschen, d.h. einen Interrupt freigeben oder wieder sperren wollen, müssen Sie folgendermaßen vorgehen.

Setzen eines Bits

Setzen Sie dazu das gewünschte Bit und zusätzlich Bit 7. Der resultierende Wert wird in das Interrupt Mask Register geschrieben. Sie wollen z.B. einen Interrupt durch eine Sprite-Sprite-Kollision erlauben (Bit 2)

```
LDA #%10000100
STA IMR
```

Sie setzen also das gewünschte Bit und zusätzlich Bit 7. Die anderen Bit (0, 1 und 3) bleiben davon unbeeinflußt.

Löschen eines Bits

Wollen Sie einen Interrupt sperren, so muß das entsprechende Bit gelöscht werden. Dazu müssen Sie das gewünschte Bit ebenfalls setzen, Bit 7 muß dabei jedoch gelöscht sein, z.B. Sperren der Sprite-Sprite-Kollision.

```
LDA #%00000100
STA IMR
```

Auch hierbei bleiben nicht gesetzte Bits unberührt. Ein Lesen des Interrupt Mask Registers ist nicht möglich. Benötigt das Programm jedoch den Wert der Interruptmaske, so können Sie ihn parallel dazu im RAM abspeichern.

Die zweite Besonderheit ist beim Interrupt Request Register zu beachten. Hat der Videocontroller einen Interrupt ausgelöst, so muß dieses Register wieder zurückgesetzt werden, da sonst beim Verlassen der Interruptroutine sofort wieder ein neuer Interrupt ausgelöst wird. Ein gesetztes Bit wird dabei ähnlich wie im Interrupt Mask Register gelöscht, einfach indem man dieses Bit wieder in das Interrupt Request Register zurückschreibt. Dies geschieht am einfachsten, indem man den Wert liest und sofort wieder zurückschreibt, z.B.

```
LDA IRR
STA IRR
```

Jetzt hat man das Bitmuster im Akku und kann durch Maskieren die einzelnen Bits testen. Dies ist immer dann erforderlich, wenn mehrere Interruptquellen aktiv sind, z.B. der normale Systeminterrupt durch den Timer und ein weiterer Interrupt durch den Videocontroller. Da beide Interrupts über den gleichen Vektor gehen, müssen wir in der Interruptroutine als erstes feststellen, wer den Interrupt ausgelöst hat und entsprechend verzweigen. Auch wenn das Ganze noch etwas kompliziert klingt, so wird ein Beispiel die Sache sicherlich verdeutlichen.

Wir wollen den Rasterinterrupt verwenden, um 16 Sprites gleichzeitig auf dem Bildschirm darzustellen. Da der

Videocontroller nur 8 Sprites gleichzeitig darstellen kann, so müssen wir durch geschicktes Umschalten zweimal je 8 Sprites nacheinander darstellen.

Das Ganze soll folgendermaßen funktionieren:

In der oberen Hälfte des Bildschirms sollen 8 Sprites dargestellt werden. Hat der Videocontroller die obere Hälfte dargestellt, so lösen wir einen Interrupt aus. In dieser Interruptroutine setzen wir nun die Parameter für die Sprites, die in der unteren Hälfte des Bildschirms dargestellt werden sollen. Gleichzeitig müssen wir den nächsten Rasterinterrupt für das Ende des Bildschirms vorbereiten, damit wir wieder auf die oberen 8 Sprites zurückschalten können.

```
PROFI-ASS 64 V2.0    SEITE 1

100:  033C                      .OPT P,00
110:             ;
120:             ; RASTERINTERRUPT
130:             ;
140:  D000       VIC      =  $D000     ; Videocontroller
160:  D001       SPRITEY  =  VIC+1     ; Sprite Y-Koordinate
165:  D012       RASTER   =  VIC+18    ; Rasterzeile
170:  D019       IRR      =  VIC+25    ; Interrupt Request Register
180:  D01A       IMR      =  VIC+26    ; Interrupt Mask Register
190:  0064       ZEILE1   =  100       ; erste Zeile
200:  00C8       ZEILE2   =  200       ; zweite Zeile
202:  005A       YCOORD1  =  90        ; erste Y-Koordinate
203:  00AA       YCOORD2  =  170       ; zweite Y-Koordinate
210:             ;
220:  0314       IRQVEC   =  $314
230:  EA31       IRQALT   =  $EA31
```

```
PROFI-ASS 64 V2.0    SEITE 2

240:                  ;
300:  033C                    *=   828
310:  033C 78         INIT    SEI
320:  033D A9 64              LDA  #ZEILE1    ; erster Interrupt
330:  033F 80 12 D0           STA  RASTER     ; bei Zeile 100
340:  0342 AD 11 D0           LDA  RASTER-1
350:  0345 29 7F              AND  #%01111111 ; Hi-Byte löschen
360:  0347 8D 11 D0           STA  RASTER-1
370:  034A A9 81              LDA  #%10000001 ; Interrupt durch
380:  034C 80 1A D0           STA  IMR        ; Rasterzeile
390:  034F A9 5B              LDA  #< TESTIRQ
400:  0351 A0 03              LDY  #> TESTIRQ
410:  0353 80 14 03           STA  IRQVEC     ; Vektor auf neue
420:  0356 8C 15 03           STY  IRQVEC+1   ; Routine
430:  0359 58                 CLI
440:  035A 60                 RTS
450:                  ;
460:  035B AD 19 D0 TESTIRQ   LDA  IRR        ; Register lesen
470:  035E 8D 19 D0           STA  IRR        ; und löschen
480:  0361 29 01              AND  #%1        ; IRQ durch Rasterzeile ?
490:  0363 D0 03              BNE  OK         ; ja
500:  0365 4C 31 EA           JMP  IRQALT     ; normaler IRQ
510:                  ;
520:  0368 AD 12 D0 OK        LDA  RASTER     ; aktuelle Zeile
530:  036B C9 C8              CMP  #ZEILE2    ; >= zweite Zeile ?
540:  036D B0 16              BCS  SECOND     ; ja
545:                  ;
550:  036F A0 C8              LDY  #ZEILE2    ; nächster IRQ bei 2. Zeile
555:  0371 A9 AA              LDA  #YCOORD2   ; neue Spritekoordinate
560:  0373 8C 12 D0 BACK      STY  RASTER     ; Rasterzeile setzen
570:  0376 A2 0E              LDX  #14
590:  0378 9D 01 D0 LOOP1     STA  SPRITEY,X  ; Spritekoordinaten
```

```
PROFI-ASS 64 V2.0    SEITE 3

      037B CA               DEX            ; ändern
      037C CA               DEX
      037D 10 F9            BPL  LOOP1
                     ;
      037F 68               PLA            ; Register zurückholen
      0380 A8               TAY
      0381 68               PLA
      0382 AA               TAX
      0383 68               PLA
      0384 40               RTI
                     ;
      0385 A0 64     SECOND LDY  #ZEILE1   ; Parameter für erste Zeile
      0387 A9 5A            LDA  #YCOORD1
      0389 4C 73 03         JMP  BACK
]033C-038C
NO ERRORS
```

Um unsere Routine zu testen, können Sie mit dem folgenden Programm 8 Sprites aktivieren. Wenn Sie nun die Interruptroutine mit SYS 828 starten, so erscheinen plötzlich 16 Sprites auf dem Bildschirm. Jeweils 8 stehen an der Y-Koordinate 90, die anderen 8 an der Y-Koordinate 170. Jedesmal wenn die oberen 8 Sprites dargestellt worden sind, verändern wir in der Interruptroutine die Spriteparameter so daß die gleichen Sprites noch einmal in der unteren Hälfte des Bildschirms vom Videocontroller dargestellt werden können.

```
100 FORI=0TO7:POKE2040+I,12:NEXT
110 V=53248
120 POKEV+21,255
130 FORI=0TO7:POKEV+2*I,(I+1)*30:POKEV+2*I+1,70:NEXT
140 FORI=0TO7:POKEV+39+I,1:NEXT
```

Außer den Spritekoordinaten können Sie natürlich auch sämtliche anderen Spriteparameter ändern, z.B. die Farbe oder die Größe. Selbstverständlich können Sie auch die Spritepointer verändern, so daß andere Spritemuster dargestellt werden können, evtl. auch in Multicolor. Aber nicht nur Sprites lassen sich verändern. Wenn Sie in der Rasterinterruptroutine den Darstellungsmodus ändern, können Sie z.B. einen geteilten Bildschirm verwenden. Die obere Hälfte des Schirms stellt eine hochauflösende Grafik dar, während Sie unten ein Textfenster haben. Wenn Sie die Zeile, bei der ein Rasterinterrupt ausgelöst wird, in einer Speicherzelle ablegen, so können Sie sogar von BASIC aus mit einer POKE-Schleife den Wert kontinuierlich ändern, so daß die Grenze wandert. Damit könnten Sie Überblendeffekte erreichen. Sie sehen, auch hier gibt es wieder ungeahnte Möglichkeiten.

2.5 Interrupt durch die CIA 6526

Nachdem wir einige Möglichkeiten der Interruptauslösung durch den Videocontroller kennengelernt haben, wollen wir uns jetzt mit der CIA 6526 beschäftigen, die sehr vielseitige Interruptquellen hat.

Die CIA 6526 ist ein universeller Ein-Ausgabe-Baustein, der zwei parallele 8-Bit-Ports, ein serielles Schieberegister, zwei 16-Bit-Timer, eine Echtzeituhr sowie mehrere Handshakeleitungen hat.

Die beiden parallelen 8-Bit-Ports dienen zur Ein- und Ausgabe von Daten. Von den insgesamt 4 Ports, die in den zwei CIAs enthalten sind, werden drei vom Betriebssystem benutzt; die beiden Ports der CIA 1 zur Abfrage der Tastaturmatrix sowie der Joysticks. Port A der CIA 2 liefert zum einen die 16K Adressauswahl für den Videocontroller (Bit 0 und 1); Bit 2 ist frei, während die Bits 3 bis 7 für den seriellen Bus benutzt werden. Port B steht als User-Port ganz dem Benutzer zur Verfügung, falls Sie nicht eine RS-232-Cartrigde zur seriellen Datenübertragung auf den User-Port gesteckt haben.

Die Timer werden vom Betriebssystem folgendermaßen genutzt:

CIA 1	Timer A	60 Hz Systeminterrupt
	Timer B	serieller Bus (Time out) Datasette lesen & schreiben
CIA 2	Timer A	RS 232 senden
	Timer B	RS 232 empfangen

Wollen Sie die Timer für eigene Zwecke benutzen, so können Sie die CIA 2 benutzen. CIA 2 liefert jedoch keinen IRQ, sondern einen NMI. Wenn Sie jedoch während Ihrer Routine nicht gleichzeitig den seriellen Bus benutzen wollen, können Sie Timer B von CIA 1 benutzen und damit auch einen IRQ auslösen. In speziellen Fällen kann man sogar auf den Systeminterrupt verzichten und Timer A mit benutzen.

Die Echtzeituhr wird vom Betriebssystem nicht genutzt; es stehen Ihnen also zwei davon zur Verfügung. Mit der Alarmzeit können Sie also wahlweise einen IRQ (CIA 1) oder einen NMI (CIA 2) auslösen.

Auch die seriellen Schieberegister können Sie frei benutzen. Die Leitung FLAG, die als Handshakeeingang dient, setzt mit einer fallenden Flanke das entsprechene Bit im Interrupt Control Register der CIA 2.

Die Ein-Ausgabe- sowie die Handshakeleitungen dienen in erster Linie zum Anschluß eigener Peripheriegeräte, ein Thema, das in dem Buch 'Der Commodore 64 und der Rest der Welt' ausführlich dargestellt ist. Dabei kommt auch die Interruptprogrammierung ausgiebig zum Einsatz. Wir werden später exemplarisch den Anschluß eines Druckers an den User-Port beschreiben; das Hauptaugenmerk soll bei uns jedoch auf der Einbindung der Routinen ins Betriebssystem liegen, so daß wir die Geräte weiterhin mit den üblichen BASIC-Befehlen OPEN, PRINT# usw. ansprechen können.

Das nächste Beispiel benutzt die Echtzeituhr und die Alarmzeit, wir wollen uns damit einen 'Wecker' programmieren. Wir benutzen dazu CIA 2, die beim Erreichen der Weckzeit einen NMI auslösen soll.

```
PROFI-ASS 64 V2.0    SEITE 1

100:  033C                 .OPT P,00
110:              ;
120:              ; WECKER MIT ECHTZEITUHR IN CIA2
130:              ;
140:  DD00        CIA2     =    $DD00     ; Basisadresse CIA
150:  DD08        TOD10    =    CIA2+8    ; Zehntel Sekunden
160:  DD09        TODSEK   =    CIA2+9    ; Sekunden
170:  DD0A        TODMIN   =    CIA2+10   ; Minuten
180:  DD0B        TODSTD   =    CIA2+11   ; Stunden
190:              ;
200:  DD0D        ICR      =    CIA2+13   ; Interrupt Control Register
210:  DD0E        CRA      =    CIA2+14   ; Control Register A
220:  DD0F        CRB      =    CIA2+15   ; Control Register B
230:  D020        BORDER   =    $D020     ; Rahmenfarbe
240:  0002        ROT      =    2
250:              ;
260:  0318        NMI      =    $318      ; NMI-Vektor
270:  FE56        CONTNMI  =    $FE56     ; alter NMI
280:              ;
290:              ; Uhrzeit  12h 00' 00.0"
300:  0000        ZEHNTEL  =    0
310:  0000        SEKUNDEN =    $00
320:  0000        MINUTEN  =    $00
330:  0012        STUNDEN  =    $12
340:              ;
350:              ; Alarmzeit  12h 00' 05.0"
360:  0000        ALARM.10 =    0
370:  0005        ALARM.SK =    $05
380:  0000        ALARM.MN =    $00
390:  0012        ALARM.ST =    $12
400:              ;
410:  033C                 *=   828
```

```
420:                    ;
430:                    ; Uhrzeit setzen
440:   033C AD 0E DD          LDA  CRA
450:   033F 09 80             ORA  #$80     ; Uhrzeittrigger 50 Hz
460:   0341 8D 0E DD          STA  CRA
470:                    ;
480:   0344 AD 0F DD          LDA  CRB
490:   0347 29 7F             AND  #$7F     ; Uhrzeit setzen
500:   0349 8D 0F DD          STA  CRB
510:                    ;
520:   034C A9 12             LDA  #STUNDEN
530:   034E 8D 0B DD          STA  TOOSTD
540:   0351 A9 00             LDA  #MINUTEN
550:   0353 8D 0A DD          STA  TODMIN
560:   0356 A9 00             LDA  #SEKUNDEN
570:   0358 8D 09 DD          STA  TODSEK
580:   035B A9 00             LDA  #ZEHNTEL
590:   035D 8D 08 DD          STA  TOD10
600:                    ;
610:   0360 AD 0F DD          LDA  CRB
620:   0363 09 80             ORA  #$80     ; Alarmzeit setzen
630:   0365 80 0F DD          STA  CRB
640:                    ;
650:   0368 A9 12             LDA  #ALARM.ST
660:   036A 8D 0B DD          STA  TODSTD
670:   0360 A9 00             LDA  #ALARM.MN
680:   036F 8D 0A DD          STA  TODMIN
690:   0372 A9 05             LDA  #ALARM.SK
700:   0374 8D 09 DD          STA  TODSEK
710:   0377 A9 00             LDA  #ALARM.10
720:   0379 8D 08 DD          STA  TOD10
730:                    ;
```

```
PROFI-ASS 64 V2.0    SEITE 3

740:   037C A9 84             LDA  #%10000100  ; Alarm
750:   037E 80 0D DD          STA  ICR       ; NMI freigeben
760:                   ;
770:   0381 A9 8C             LDA  #< TEST
780:   0383 A0 03             LDY  #> TEST
790:   0385 8D 18 03          STA  NMI          ; neuer NMI-Vektor
800:   0388 8C 19 03          STY  NMI+1
810:   038B 60                RTS
820:                   ;
830:   038C 48         TEST   PHA
840:   0380 8A                TXA
850:   038E 48                PHA               ; Register retten
860:   038F 98                TYA
870:   0390 48                PHA
880:   0391 AC 0D DD          LDY  ICR
880:   0394 98                TYA
890:   0395 29 04             AND  #%100        ; Alarmbit gesetzt ?
900:   0397 D0 03             BNE  ALARM        ; ja
910:   0399 4C 56 FE          JMP  CONTNMI
920:                   ;
930:   039C A9 02      ALARM  LDA  #ROT
940:   039E 8D 20 D0          STA  BORDER       ; Rahmenfarbe auf Rot
950:                   ;
960:   03A1 68                PLA
960:   03A2 A8                TAY
970:   03A3 68                PLA
970:   03A4 AA                TAX
980:   03A5 68                PLA
980:   03A6 40                RTI
]033C-03A7
NO ERRORS
```

Das Programm definiert zuerst die Adressen der Echtzeituhr und der Kontrollregister in der CIA 2. Dann werden die Uhrzeit auf 12 Uhr und die Alarmzeit auf 12 Uhr und 5 Sekunden gesetzt. Das Programm setzt als erstes den Uhrzeittrigger auf 50 Hz, damit die Uhr korrekt läuft. Dann wird im Control Register B Bit 7 gelöscht, um der CIA anzuzeigen, daß wir die Uhrzeit eingeben wollen, was anschließend geschieht. Jetzt setzen wir Bit 7, programmieren die Alarmzeit und geben im Interrupt Control Register den NMI für die Alarmzeit frei. Dazu müssen Bit 2 sowie Bit 7 gesetzt werden. Abschließend wird noch der NMI-Vektor auf unsere neue Routine gesetzt und die Initialisierung ist beendet.

Die eigentliche NMI-Routine hat nicht mehr viel zu tun. Zuerst werden die Register auf den Stack gerettet, dann wird das Interrupt Control Register gelesen und Bit 2 geprüft. War das Bit gesetzt, so ist die Alarmzeit erreicht. Wir reagieren darauf, indem wir die Rahmenfarbe des Bildschirms auf Rot setzen. Nun werden die Register wieder zurückgeholt und mit RTI kann ins unterbrochene Programm zurückgekehrt werden. Wurde der NMI nicht durch die Alarmzeit ausgelöst, so springen wir in die NMI-Routine des Betriebssystems. Dort wird geprüft, ob neben der RESTORE-Taste (die den NMI ausgelöst hat) noch die STOP-Taste gedrückt war. Ist dieser Test positiv, so wird ein Warmstart durchgeführt.

Die Aktion auf das Erreichen der Alarmzeit hin bleibt natürlich Ihnen überlassen; Sie können z.B. das Programm als Wecker benutzen, indem Sie dem Sound-Chip irgendwelche Töne entlocken. Allerdings sollten Sie dann bei der Aktivierung der Routine eine komfortablere Möglichkeit zur Eingabe von Uhr- und Alarmzeit vorsehen. Die Echtzeituhr hat übrigens eine sehr hohe Langzeitkonstanz, da sie netzsynchron läuft.

2.6 Die Benutzung der Timer

Jede CIA enthält zwei 16-Bit Timer, nach deren Ablauf ein Interrupt ausgelöst werden kann. Diese Timer werden vom Betriebssystem ausgiebig verwendet. Die Timer werden bei jedem Prozessortakt um eins heruntergezählt. Wird der Wert Null erreicht, so wird das entsprechende Bit im Interrupt Control Register gesetzt und - falls durch die Maske im Interrupt Control Register erlaubt - ein IRQ oder NMI ausgelöst. Da der Commodore 64 in der deutschen PAL-Version mit einer Taktfrequenz von ca. 985 kHz läuft, ist ein Taktzyklus 1.015 Mikrosekunden lang oder abgerundet eine Mikrosekunde. Da die Timer mit einem 16-Bit-Wert geladen werden können, lassen sich so Zeiten bis zu 65535 Taktzyklen, das sind ca. 65 ms oder etwa eine fünfzehntel Sekunde erreichen. Timer A der CIA 1 wird z.B. mit dem Wert $4025 gleich 16421 Taktzyklen geladen, was einer sechzigstel Sekunde entspricht. Bei der amerikanischen NTSC-Version beträgt die Taktfrequenz 1.02 MHz. Dort wird der Timer mit $4295 gleich 17045 geladen, was bei der etwas höheren Taktfrequenz ebenfalls einer sechzigstel Sekunde entspricht.

Für die Timer gibt es nun verschiedene Betriebsarten, z.B. unterscheidet man den 'one shot' und den 'continuous' Mode. Beim one shot mode zählt der Timer nur einmal von Ausgangwert auf null und bleibt dann stehen; im continuous mode wird der Timer danach automatisch wieder mit dem Startwert geladen und neu gestartet. Neben dem Auslösen eines Interrupts können die Timer beim Nulldurchgang auch einen Impuls am Userport auslösen. Damit kann z.B. ein Taktsignal für ein Peripheriegerät erzeugt werden. Die Timer können außerdem noch als Zähler eingesetzt werden. Dabei werden sie nicht durch den Systemtakt dekrementiert, sondern durch von außen

angelegte Signale. Weiterhin kann man die Timer koppeln. Dabei zählt ein Timer immer dann weiter, wenn der andere Timer Null erreicht hat. Dadurch hat man praktisch einen 32-Bit-Timer, so daß Zeiten bis zu 2^32 Taktzyklen erreicht werden können, das sind 4 294 967 296 Zyklen oder ca. 4360 Sekunden bzw. 1 Std. und 12 Min.

Zum Abschluß unseres Kapitels über die Interruptprogrammierung wollen wir nun ein Maschinenprogramm schreiben, das es uns erlaubt, auch im BASIC Unterprogramme interruptgesteuert ablaufen zu lassen. Dabei werden wir sowohl etwas über die Benutzung der Timer als auch die Arbeit des BASIC-Interpreters lernen.

Dazu führen wir einen neuen BASIC-Befehl ein, der es uns erlaubt, ein normales BASIC-Unterprogramm immer dann ausführen zu lassen, wenn eine bestimmte Zeit vergangen ist. Vorweg jedoch ein klein wenig Theorie.

Der BASIC-Interpreter befindet sich bei der Abarbeitung eines BASIC-Programms in der sogenannten Hauptschleife, in der er jede Anweisung analysiert und ausführt. Nach jedem Statement prüft er, ob die Stoptaste gedrückt wurde. Hat er diese Taste erkannt, so verläßt er die Hauptschleife und kehrt in den Direktmodus zurück. Das Abfragen der Stoptaste geschieht über einen Sprungvektor. Diesen Vektor verändern wir nun so, daß er auf eine neue Routine zeigt. In dieser Routine wird nun geprüft, ob die Bedingung zur Ausführung des Interruptprogramms erfüllt ist - mit anderen Worten ob unser Timer schon abgelaufen ist. Um dies zu erkennen, wird von einer echten Interruptroutine nach Ablauf des Timers ein Flag gesetzt, das von der obigen Routine getestet werden kann.

Der neue BASIC-Befehl sagt nun, welche BASIC-Routine nach einem Interrupt ausgeführt werden soll. Als zweiten Parameter geben wir noch an, nach welcher Zeit ein Interrupt ausgelöst werden soll. Der Befehl sieht dann so aus.

```
!GOSUB 1000,100
```

Dabei dient das Ausrufungszeichen zur Unterscheidung vom normalen GOSUB-Befehl. Die 1000 ist wie üblich die Zeilennummer des Unterprogramms und die 100 gibt die Zeit an, nach der ein Interrupt ausgelöst werden soll. Als Zeiteinheit wählen wir dabei eine fünfzigstel Sekunde. Mit diesem Wert laden wir einen Timer. Den zweiten Wert laden wir in den nächsten Timer, die wir zusammen als 32-Bit-Timer betreiben. Wir können dann Zeiten von einer fünfzigstel bis zu 65535 fünfzigstel Sekunden programmieren, das sind 0.02 bis 1311 Sekunden oder 21 Minuten und 51 Sekunden.

Unser Programm besteht also außer der Initialisierung aus drei Routinen. Die erste modifiziert den BASIC-Interpreter dahingehend, daß er unseren neuen Befehl versteht. Die zweite Routine prüft nach jedem Statement, ob das Flag für den abgelaufenen Timer schon gesetzt ist und verzweigt, falls erforderlich, in die BASIC-Subroutine. Das dritte Programm schließlich ist unsere Interrupt- bzw. NMI-Routine, die nach Ablauf der Timer das Flag für die zweite Routine setzt.

```
100:   CC00                  .OPT P,00
110:   CC00                  .TIT "BASIC-IRQ"
120:   CC00                  .SYM          ; Symboltabelle ausgeben
130:             ;
140:             ; Interruptroutine für BASIC
150:             ;
160:   0308      EXEC     =   $308      ; Vektor für Statement ausführen
170:   0318      NMI      =   $318      ; NMI-Vektor
180:   0328      STOP     =   $328      ; STOP-Vektor
190:             ;
200:   DD00      CIA2     =   $0D00
210:   DD04      TIMERA   =   CIA2+4    ; Timer A
220:   DD06      TIMERB   =   CIA2+6    ; Timer B
230:   DD0D      ICR      =   CIA2+13   ; Interrupt Control Register
240:   DD0E      CRA      =   CIA2+14   ; Control Register A
250:   DD0F      CRB      =   CIA2+15   ; Control Register B
260:             ;
270:   FE56      CONTNMI  =   $FE56     ; alten NMI fortsetzen
280:             ;
290:   4CF9      TIME     =   19705     ; = 20 Millisekunden
300:   0014      LO       =   $14       ; Zeilennummer lo
310:   0015      HI       =   LO+1
320:   005F      LINEADR  =   $5F       ; Adresse der BASIC-Zeile
330:   0039      LINENO   =   $39       ; laufende Zeilennummer
340:   0073      CHRGET   =   $73
350:   0079      CHRGOT   =   CHRGET+6
360:   007A      TXTPTR   =   CHRGOT+1
370:   008D      GOSUB    =   $80       ; GOSUB-Token
380:   AF08      SYNTAX   =   $AF08     ; SYNTAX ERROR
390:   A8E3      UNDEFD   =   $A8E3     ; UNDEF'D STATEMENT ERROR
400:   B248      ILLQUAN  =   $B248     ; ILLEGAL QUANTITY ERROR
410:   A7AE      INTER    =   $A7AE     ; Interpreterschleife
```

```
420:  A96B            GETLIN    =    $A96B    ; Zeilenummer holen
430:  A613            GETADR    =    $A613    ; Zeile suchen
440:  AEFD            CHKCOM    =    $AEFD    ; Komma testen
450:  A7E7            EXECOLD   =    $A7E7    ; Statement ausführen
460:  AD8A            FRMNUM    =    $AD8A    ; numerischen Wert holen
470:  B7F7            INTEGER   =    $B7F7    ; und nach INTEGER wandeln
480:  A3FB            TESTSTACK=     $A3FB    ; Prüfung auf Platz im Stack
490:  F6ED            TESTOLD   =    $F6ED    ; STOP-Taste prüfen
500:  FE47            NMIOLD    =    $FE47    ; alter NMI-Vektor
510:                  ;
520:  CC00                      *=   $CC00
530:  CC00 A9 10      INIT      LDA  #< TESTSTAT
540:  CC02 A0 CC                LDY  #> TESTSTAT
550:  CC04 80 08 03             STA  EXEC     ; Routine zur Dekodierung
560:  CC07 8C 09 03             STY  EXEC+1   ; von '!' einbinden
570:  CC0A A9 00                LDA  #0
580:  CC0C 80 F7 CC             STA  FLAG     ; Flag löschen
590:  CC0F 60                   RTS
600:                  ;
610:  CC10 20 73 00 TESTSTAT JSR  CHRGET   ; nächstes Zeichen holen
620:  CC13 C9 21                CMP  #"!"
630:  CC15 F0 06                BEQ  TSTGOSUB
640:  CC17 20 79 00             JSR  CHRGOT   ; Flags wieder herstellen
650:  CC1A 4C E7 A7             JMP  EXECOLD   ; und normal weiter
660:                  ;
670:  CC1D 20 73 00 TSTGOSUB JSR  CHRGET   ; nächstes Zeichen
680:  CC20 C9 80                CMP  #GOSUB   ; GOSUB-Kode ?
690:  CC22 F0 03                BEQ  OK       ; ja
700:  CC24 4C 08 AF             JMP  SYNTAX   ; SYNTAX ERROR
710:  CC27 20 73 00 OK          JSR  CHRGET   ; nächstes Zeichen
720:  CC2A F0 68                BEQ  IRQOFF   ; Zeilenende, dann IRQ abschalten
730:  CC2C 20 6B A9             JSR  GETLIN   ; Zeilennummer holen
```

```
CC2F 20 13 A6           JSR  GETADR  ; Adresse der Zeile holen
CC32 B0 03              BCS  FOUND   ; gefunden ?
CC34 4C E3 A8           JMP  UNDEFD  ; nein, UNDEF'D STATEMENT ERROR
CC37 A5 5F     FOUND    LDA  LINEADR ; Zeilenadresse
CC39 E9 01              SBC  #1      ; minus 1
CC3B 8D F8 CC           STA  LINESTR  ; merken
CC3E A5 60              LDA  LINEADR+1
CC40 E9 00              SBC  #0      ; Hi-Byte
CC42 8D F9 CC           STA  LINESTR+1
CC45 20 FD AE           JSR  CHKCOM  ; auf Komma prüfen
CC48 20 8A AD           JSR  FRMNUM  ; Wert holen
CC4B 20 F7 B7           JSR  INTEGER  ; nach INTEGER wandeln
CC4E A5 14              LDA  LO
CC50 05 15              ORA  HI      ; Lo- und Hi-Byte null ?
CC52 D0 03              BNE  OK1     ; nein
CC54 4C 48 B2           JMP  ILLQUAN  ; ILLEGAL QUANTITY ERROR
CC57 A5 15     OK1      LDA  HI
CC59 8D 07 DD           STA  TIMERB+1
CC5C A5 14              LDA  LO      ; Wert in Timer B laden
CC5E 8D 06 DD           STA  TIMERB
CC61 A9 4C              LDA  #> TIME ; Timer A mit
CC63 8D 05 DD           STA  TIMERA+1
CC66 A9 F9              LDA  #< TIME ; 20 ms laden
CC68 8D 04 DD           STA  TIMERA
CC6B A9 11              LDA  #%00010001  ; Timer A starten
CC60 80 0E DD           STA  CRA
CC70 A9 51              LDA  #%01010001  ; Timer B starten
CC72 8D 0F DD           STA  CRB     ; durch Timer A getriggert
CC75 AD 0D DD           LDA  ICR     ; ICR löschen
CC78 A9 82              LDA  #%10000010  ; NMI für Timer B
CC7A 8D 0D DD           STA  ICR     ; freigeben
CC7D A9 C9              LDA  #< TESTTIME
```

```
1060:  CC7F A0 CC                LDY  #> TESTTIME
1070:  CC81 80 28 03             STA  STOP     ; STOP-Vektor setzen
1080:  CC84 8C 29 03             STY  STOP+1
1090:  CC87 A9 B0                LDA  #< NMIROUT
1100:  CC89 A0 CC                LDY  #> NMIROUT
1110:  CC8B 80 18 03             STA  NMI      ; NMI-Vektor setzen
1120:  CC8E 8C 19 03             STY  NMI+1
1130:  CC91 4C AE A7             JMP  INTER    ; zur Interpreterschleife
1140:                  ;
1150:  CC94 A9 7F      IRQOFF    LDA  #%01111111
1160:  CC96 8D 0D DD             STA  ICR      ; alle Interrupts aus
1170:  CC99 A9 ED                LDA  #< TESTOLD
1180:  CC9B A0 F6                LDY  #> TESTOLD
1190:  CC90 80 28 03             STA  STOP     ; STOP-Vektor auf alten Wert
1200:  CCA0 8C 29 03             STY  STOP+1
1210:  CCA3 A9 47                LDA  #< NMIOLD
1220:  CCA5 A0 FE                LDY  #> NMIOLD
1230:  CCA7 80 18 03             STA  NMI      ; NMI-Vektor auf alten Wert
1240:  CCAA 8C 19 03             STY  NMI+1
1250:  CCA0 4C AE A7             JMP  INTER    ; zur Interpreterschleife
1260:                  ;
1270:  CCB0 48         NMIROUT   PHA
1280:  CCB1 8A                   TXA
1280:  CCB2 48                   PHA
1290:  CCB3 98                   TYA
1290:  CCB4 48                   PHA
1300:  CCB5 AC 0D DD             LDY  ICR
1310:  CCB8 98                   TYA
1320:  CCB9 29 02                AND  #%10     ; Timer B abgelaufen ?
1330:  CCBB D0 03                BNE  TIMEOUT  ; ja
1340:  CCBD 4C 56 FE             JMP  CONTNMI  ; sonst normaler NMI
1350:                  ;
```

```
1360:  CCC0 EE F7 CC TIMEOUT  INC  FLAG     ; Flag setzen
1370:  CCC3 68                PLA
1370:  CCC4 A8                TAY
1380:  CCC5 68                PLA
1380:  CCC6 AA                TAX
1390:  CCC7 68                PLA
1390:  CCC8 40                RTI
1400:                 ;
1410:  CCC9 AD F7 CC TESTTIME LDA  FLAG     ; Flag gesetzt ?
1420:  CCCC D0 03             BNE  TIMEIRQ  ; ja
1430:  CCCE 4C ED F6          JMP  TESTOLD
1440:                 ;
1450:  CCD1 CE F7 CC TIMEIRQ  DEC  FLAG     ; Flag wieder löschen
1460:  CCD4 68                PLA
1460:  CCD5 68                PLA           ; Rücksprungadresse vom Stack
1470:  CCD6 A9 03             LDA  #3
1480:  CC08 20 FB A3          JSR  TESTSTACK  ; noch genügend Platz im Stack?
1490:  CCDB A5 7B             LDA  TXTPTR+1
1500:  CCDD 48                PHA           ; CHRGET-Pointer auf Stack
1510:  CCDE A5 7A             LDA  TXTPTR
1520:  CCE0 48                PHA
1530:  CCE1 A5 3A             LDA  LINENO+1
1540:  CCE3 48                PHA           ; aktuelle Zeilennummer auf Stack
1550:  CCE4 A5 39             LDA  LINENO
1560:  CCE6 48                PHA
1570:  CCE7 A9 8D             LDA  #GOSUB
1580:  CCE9 48                PHA           ; GOSUB-Kode auf Stack
1590:  CCEA AD F8 CC          LDA  LINESTR
1600:  CCED 85 7A             STA  TXTPTR   ; Adresse der Subroutine
1610:  CCEF AD F9 CC          LDA  LINESTR+1
1620:  CCF2 85 7B             STA  TXTPTR+1
1630:  CCF4 4C B1 A7          JMP  INTER+3  ; zur Interpreterschleife
```

```
BASIC-IRQ      PROFI-ASS 64 V2.0     SEITE 6

1640:                  ;
1650:  CCF7            FLAG      *=    *+1
1660:  CCF8            LINESTR   *=    *+2
]CC00-CCFA
NO ERRORS

BASIC-IRQ      PROFI-ASS 64 V2.0     SEITE 7

SYMBOLTABLE:
LINESTR   CCF8  FLAG      CCF7  TIMEIRQ   CCD1  TESTTIME  CCC9
TIMEOUT   CCC0  NMIROUT   CCB0  IRQOFF    CC94  OK1       CC57
FOUND     CC37  OK        CC27  TSTGOSUB  CC1D  TESTSTAT  CC10
INIT      CC00  NMIOLD    FE47  TESTOLD   F6ED  TESTSTAC  A3FB
INTEGER   B7F7  FRMNUM    AD8A  EXECOLD   A7E7  CHKCOM    AEFD
GETADR    A613  GETLIN    A96B  INTER     A7AE  ILLQUAN   B248
UNDEFD    A8E3  SYNTAX    AF08  GOSUB     008D  TXTPTR    007A
CHRGOT    0079  CHRGET    0073  LINENO    0039  LINEADR   005F
HI        0015  LO        0014  TIME      4CF9  CONTNMI   FE56
CRB       DD0F  CRA       DD0E  ICR       DD0D  TIMERB    DD06
TIMERA    DD04  CIA2      DD00  STOP      0328  NMI       0318
EXEC      0308
45 SYMBOLS DEFINED
```

Bevor wir nun zur ausführlichen Besprechung des Programms kommen, hier ein erstes Demonstrationsprogramm.

```
100 SYS 52224:REM ERWEITERUNG INITIALISIEREN
110 !GOSUB 200,50
120 I=I+1 : PRINT I : IF I<100 GOTO 120
130 !GOSUB
140 END
200 J=J+1 : PRINT J ". IRQ-AUFRUF !" : RETURN
```

Wenn Sie dieses Programm mit RUN starten, so wird in Zeile 100 unsere Befehlserweiterung initialisiert. Zeile 110 definiert das Unterprogramm ab Zeile 200 als Interruptprogramm, das jede Sekunde (50 Fünfzigstel) ausgeführt werden soll. Das eigentliche Hauptprogramm steht in Zeile 120 und gibt die Zahlen von 1 bis 100 aus. Ist diese Schleife beendet, so wird die Interruptroutine durch !GOSUB ohne Parameter wieder abgeschaltet und das Programm beendet. Die Interruptroutine steht in Zeile 200. Sie gibt bei jedem Aufruf einen fortlaufenden Zähler mit einem Text aus bevor mit RETURN wieder ins unterbrochene Hauptprogramm zurückgekehrt wird.

Wenn Sie das Programm laufen lassen, so werden die Zahlen von 1 bis 100 ausgegegben, jedoch fünfmal unterbrochen durch die Ausgabe

```
1 . IRQ-AUFRUF !
```

bis

```
5 . IRQ-AUFRUF !
```

Wenn Sie den zweiten Parameter in Zeile 110 verändern, so können Sie die Aufruffrequenz bestimmen. Es sind Werte zwischen 1 und 65535 zugelassen. Je kleiner der Wert wird, desto häufiger wird die Interruptroutine aufgerufen. Dabei darf die Zeit für das Ausführen der BASIC-Interruptroutine jedoch nicht länger sein als die Zeit zwischen zwei Aufrufen, da die Interruptroutine sonst durch sich selbst unterbrochen wird und der BASIC-Stack überläuft. Mit

```
110 !GOSUB 200,1
```

erhalten Sie folgende Ausgabe:

```
1
1 . IRQ-AUFRUF !
2 . IRQ-AUFRUF !
....
22 . IRQ-AUFRUF !
23 . IRQ-AUFRUF !

?OUT OF MEMORY ERROR IN 200
```

Je nach Länge der Interruptroutine darf also eine maximale Aufruffrequenz nicht überschritten werden. Doch nun zur Besprechung des Maschinenprogramms.

Von Zeile 100 bis 500 werden Konstanten definiert. Dies betrifft NMI- und BASIC-Vektoren. Dann folgen die Register in der CIA2, die für den Timerinterrupt benötigt werden. Zeile 290 definiert unsere Zeiteinheit. Dann kommen BASIC-Adressen aus der Zeropage, sowie Fehlermeldungen und ROM-Adressen des BASIC-Interpreters. Von Zeile 520 bis 590 erfolgt die Initialisierung. Hierbei wird der Vektor, der auf die Routine zur Dekodierung und Ausführung eines BASIC-Statements zeigt, auf unsere eigene Routine 'umgebogen'. Diese Routine holt sich das nächste Zeichen aus dem BASIC-Text und vergleicht es mit dem Ausrufungszeichen. Haben wir dieses Zeichen nicht gefunden, so werden mit der CHRGOT-Routine die ursprünglichen Werte der Flags wieder hergestellt und an die Stelle im Interpreter gesprungen, an der normalerweise die Statements verarbeitet werden. Haben wir jedoch ein Ausrufungszeichen gefunden, so holen wir uns das nächste Zeichen und prüfen, ob es der Kode für GOSUB ist. Falls nicht, so springen wir zur Ausgabe von 'SYNTAX ERROR'. Das nächste Zeichen wird geholt. Handelt es sich um das Zeilenende, so wird zu der Routine

verzweigt, die den Interrupt wieder abstellt und die Vektoren auf den alten Wert zurücksetzt. Nun wird die Zeilennummer sowie deren Adresse geholt. Nachdem geprüft wurde, ob diese Zeile überhaupt existiert (durch das gesetzte Carry-Flag signalisiert), wird die Zeilenadresse um eins vermindert und für später gespeichert. Jetzt kann auf das nächste Komma getestet sowie der zweite Parameter geholt werden. Nachdem wir uns vergewissert haben, daß er nicht null ist, wird damit Timer B geladen. Timer A wird nun mit dem Wert für eine fünfzigstel Sekunde geladen und beide Timer werden gestartet. Dabei wird Timer B so programmiert, daß er bei jedem Unterlauf von Timer A dekrementiert wird. Anschließend wird der NMI für Timer B durch Einschreiben des entsprechenden Bitmusters in das Interrupt Control Register freigegeben. Zum Schluß werden noch STOP- und NMI-Vektor auf die nachfolgenden neuen Routinen gesetzt, ehe wir in die Interpreterschleife zurückkehren.

Von Zeile 1150 bis 1250 finden Sie die Routine, die nach einem !GOSUB-Befehl ohne Parameter die Interrupts abschaltet und die Vektoren wieder auf die alten Werte setzt. Die eigentliche NMI-Routine steht von Zeile 1270 bis 1390. Es werden wie üblich zuerst die Register gerettet und dann durch Abfrage des Interrupt Control Registers getestet, ob Timer B den NMI ausgelöst hat. War das der Fall, so wird ein Flag gesetzt und aus der NMI-Routine zurückgekehrt. Ansonsten wird in die Standard-NMI-Routine verzweigt.

Das wichtigste Unterprogramm, daß vom BASIC-Interpreter nach jedem Statement aufgerufen wird, finden Sie ab Zeile 1410. Hier wird nun geprüft, ob die Zeit bereits abgelaufen ist und das entsprechende Flag von der NMI-Routine gesetzt wurde. Verläuft der Test negativ, so wird in die normale Routine zum Test der Stoptaste gesprungen. War die Zeit jedoch

abgelaufen, wird als erstes das Flag wieder gelöscht und dann die eigene Rücksprungadresse vom Stack genommen. Nun wird das nachvollzogen, was der BASIC-Interpreter bei einem GOSUB-Befehl macht. Nachdem sichergestellt ist, daß noch genügend Platz auf dem Stack ist, werden der Zeiger in den BASIC-Text sowie die aktuelle Zeilennummer auf den Stack gebracht. Zur Unterscheidung von einer FOR-NEXT-Schleife, deren Parameter ebenfalls auf dem Stack abgelegt werden, kommt noch der GOSUB-Kode auf den Stack. Nun wird die bei der Definition berechnete und abgespeicherte Adresse des Unterprogramms in den BASIC-Textzeiger geladen und wieder zur Interpreterschleife verzweigt. Der BASIC-Interpreter führt nun das Unterprogramm aus und kann beim RETURN-Befehl wieder korrekt in das unterbrochene Programm zurückkehren.

Das Programm endet mit der Definition von zwei Variablen. Durch den .SYM-Befehl in Zeile 120 erhalten Sie die Symboltabelle mit allen verwendeten Symbolen und ihren Werten.

Mit dieser Befehlserweiterung stehen Ihnen in BASIC nun Möglichkeiten offen, die sonst der Maschinensprache vorbehalten waren. Sie können nun timergesteuert Unterprogramme in BASIC aufrufen. Dabei haben Sie die weite Zeitspanne zwischen 20 Millisekunden und 21 Minuten zur Verfügung. Wenn Sie also einmal ein Maschinenprogramm geladen haben, so können Sie fortan timergesteuerte Interruptroutinen in BASIC schreiben. Als Beispiel haben wir ein Programm, das den Bildschirm blinken läßt, indem es Hintergrund- und Rahmenfarbe vertauscht.

```
100 SYS 52224
110 F1 = 53280 : F2 = F1 + 1
120 !GOSUB 1000,30
```

```
130 FOR I=1 TO 1000 : PRINT I, : NEXT
140 !GOSUB : END
1000 A=PEEK(F1) : POKE F1,PEEK(F2) : POKE F2,A : RETURN
```

Das Abschalten der BASIC-Interruptroutine mit !GOSUB sollte immer vor Beendigung des Programms geschehen. Wenn Sie später z.B. bei aktivierter Interruptroutine ein Programm listen oder abspeichern wollen, so wird dies direkt durch unsere Interruptroutine unterbrochen, da bei diesen Befehlen ständig die Stoptaste abgefragt wird.

Das nächste Beispielprogramm gibt den Zeichensatz des Commdore 64 in Normal- und Reversdarstellung aus und schaltet dann per Interrupt zwischen der Standardtextdarstellung und dem Extended Color Mode um. Dies geschieht durch Setzen von Bit 6 im Register 17 des Videocontrollers. In diesem Modus können nicht mehr 256, sondern nur noch 64 verschiedene Zeichen dargestellt werden. Die frei werdenden obersten beiden Bits des Bildschirmkodes dienen nun zur Auswahl von vier verschiedenen Hintergrundfarben für ein Zeichen, die in den Registern 33 bis 36 des Videocontrollers abgelegt sind (Adressen 53281 bis 53284).

```
100 SYS 52224
110 !GOSUB 170,25
120 X=18
130 PRINTCHR$(X);:X=X+128AND255
140 FORI=32TO 127:PRINTCHR$(I);:NEXT
150 FORI=160TO 255:PRINTCHR$(I);:NEXT
160 PRINT:GOTO130
170 A=PEEK(53248+17):POKE53248+17,(AOR64)ANDNOT(AAND64)
180 RETURN
```

3.1 Betriebssystem- und BASIC-Erweiterungen

Der Commodore 64 hat gegenüber seinen "großen" Brüdern aus den CBM-Reihen 2000, 3000, 4000 und 8000 den Vorteil, daß man den BASIC-Interpreter und das Betriebssystem dieses Rechners auf einfachste Weise um eigene Routinen erweitern kann, die man in BASIC-Interpreter und Betriebssystem 'einbinden' kann.

Mit 'Einbinden' ist gemeint, daß nach der Initialisierung dieser Routinen die erweiterten Möglichkeiten durch neue oder erweiterte Befehlsworte genutzt werden können. Es ist dann also nicht mehr erforderlich, jeden neuen Befehl mit PEEK, POKE oder SYS anzusprechen. Zur Realisierung dieses Verfahrens gibt es zwei grundsätzliche Möglichkeiten.

Da der komplette Adressraum des Commodore 64 von 64 KByte mit RAM ausgestattet ist, kann man Änderungen im BASIC und Betriebsystem einfach dadurch vornehmen, daß man das BASIC und/oder das Betriebsystem aus dem ROM einfach in das an der gleichen Adresse liegende RAM kopiert, dort die gewünschten Änderungen vornimmt und dann einfach mittels des Prozessorports an der Adresse 1 auf das RAM 'umschaltet'. Diese Methode hat gegenüber der noch zu schildernden Methode sowohl Vor- als auch Nachteile.

Der Vorteil dieser Methode besteht darin, daß man völlige Freiheit bei der Modifikation hat. Das geht soweit, daß anstelle des BASIC eine komplette andere Sprache verwendet werden kann oder ein komplett neues Betriebssystem. Diese RAM-Bereiche werden sonst z.B. oft für Grafikdarstellungen benutzt. Der Nachteil dieser Methode liegt darin, daß dieser RAM-Bereich damit nicht mehr zur Verfügung steht. Eine Variante dieser Methode besteht darin, daß man in den

Adressbereich von $8000 bis $9FFF oder von $8000 bis $BFFF ein oder zwei EPROMs legt, die eine BASIC-Erweiterung, eine andere Sprache oder ein anwenderspezifisches Programm enthalten. Dazu ist jedoch eine Karte erforderlich, die in den Expansionport gesteckt wird.

Die zweite Methode benötigt nicht das zusätzliche ROM, sondern benutzt schon vorgesehene Einsprungpunkte, um die wichtigsten Funktionen zu modifizieren. Dazu wird an diesen Schlüsselpositionen mit sogenannten Sprungvektoren gearbeitet, die vom Anwender geändert werden können. An diesen Stellen wird der indirekte Sprungbefehl verwendet, z.B.

JMP (VEKTOR)

An der Adresse VEKTOR sind nun Lo- und Hi-Byte der eigentlichen Sprungadresse gespeichert. Diese Vektoren werden beim Einschalten des Rechners initialisiert und zeigen beim BASIC-Interpreter meist direkt hinter den indirekten Sprungbefehl. Wollen wir nun eine bestimmte Funktion ändern, so schreiben wir dazu eine eigene Routine und ändern den zugehörigen Sprungvektor so, daß er auf unsere neue Funktion zeigt. Das Prinzip ist also ähnlich wie wir es schon bei den Interruptvektoren kennengelernt haben.

Bei Benutzung der 'RAM-Methode' gibt Ihnen die folgende Tabelle Auskunft darüber, welches Bitmuster Sie in Adresse eins schreiben müssen, damit Sie die gewünschte Speicherkonfiguration erhalten.

Bit						
2	1	0	dez.	$A000 - $BFFF	$D000 - $DFFF	$E000 - $FFFF
1	1	1	7	BASIC	I/O	KERNAL
1	1	0	6	RAM	I/O	KERNAL
1	0	1	5	RAM	I/O	RAM
1	0	0	4	RAM	RAM	RAM
0	1	1	3	BASIC	CHAR-GEN	KERNAL
0	1	0	2	RAM	CHAR-GEN	KERNAL
0	0	1	1	RAM	CHAR-GEN	RAM
0	0	0	0	RAM	RAM	RAM

Die Tabelle enthält alle möglichen Kombinationen für die Speicherkonfiguration, die per Programm eingestellt werden können. Dabei ergeben die Kombinationen 4 und 7 das gleiche Ergebnis; der komplette Adressraum ist auf RAM geschaltet. Aus der Tabelle geht hervor, daß zwar das BASIC-ROM alleine gegen RAM ausgetauscht werden kann, das Kernal-ROM läßt sich jedoch nur zusammen mit dem BASIC gegen RAM austauschen. Dies ist zu beachten, wenn das Betriebssystem ersetzt werden soll. Der Adressbereich $D000 - $DFFF ist dreifach belegt: Einmal der I/O-Bereich, der sich wie folgt aufgliedert:

$0000 - $D3FF	VIC 6569
$D400 - $07FF	SID 6581
$0800 - $DBFF	Farbram
$DC00 - $DCFF	CIA 1 6526
$DD00 - $DDFF	CIA 2 6526
$DE00 - $DEFF	I/O 1 für Erweiterungen
$DF00 - $DFFF	I/O 2 für Erweiterungen

Desweiteren kann an dieser Adresse der Charctergenerator angesprochen werden. Zum dritten ist dieser Bereich ebenfalls mit RAM belegt, das sich allerdings nur dann ansprechen läßt, wenn der komplette Speicher auf RAM geschaltet ist.

3.2 Die BASIC-Vektoren

Der BASIC-Interpreter hat sechs Vektoren, die an den wichtigsten Stellen eine Einbindung von eigenen Routinen ermöglichen. Diese Vektoren sind in Page 3 abgelegt und haben folgende Bedeutung:

Vektor	Adresse	Bedeutung
$0300/$0301	$E38B	BASIC-Warmstart und Fehlereinsprung
$0302/$0303	$A483	Eingabe-Warteschleife
$0304/$0305	$A57C	Umwandlung in Interpreterkode
$0306/$0307	$A71A	Interpreterkode in Klartext wandeln
$0308/$0309	$A7E4	BASIC-Befehl ausführen
$030A/$030B	$AE86	BASIC-Ausdruck auswerten

Mit Hilfe dieser 6 Vektoren hat man große Einflußmöglichkeiten auf den BASIC-Interpreter; man kann damit eigene Befehle und Funktionen hinzufügen. Wir werden die Bedeutung jedes Vektors kennenlernen und für Erweiterungen nutzen.

Damit Sie den größten Nutzen aus diesem Kapitel ziehen können, sollten Sie parallel zur Durcharbeitung dieses Kapitels das ROM-Listing des Commodore 64, wie Sie es in '64 intern' finden, zur Hand haben. Dadurch können wir uns auf die entsprechenden Routinen des BASIC-Interpreters beziehen.

Der Warmstart- und Fehlervektor $300/$301

Über diesen Vektor wird nach END sowie nach dem Auftreten eines Fehlers gesprungen. Ist ein Fehler aufgetreten, so muß das X-Register die Fehlernummer enthalten. Diese Nummern gehen von eins bis 29 und haben folgende Bedeutung:

Nr.	Fehlermeldung
1	TOO MANY FILES
2	FILE OPEN
3	FILE NOT OPEN
4	FILE NOT FOUND
5	DEVICE NOT PRESENT
6	NOT INPUT FILE
7	NOT OUTPUT FILE
8	MISSING FILENAME
9	ILLEGAL DEVICE NUMBER
10	NEXT WITHOUT FOR
11	SYNTAX
12	RETURN WITHOUT GOSUB
13	OUT OF DATA
14	ILLEGAL QUANTITY
15	OVERFLOW
16	OUT OF MEMORY
17	UNDEF'D STATEMENT
18	BAD SUBSCRIPT
19	REDIM'D ARRAY
20	DIVISION BY ZERO
21	ILLEGAL DIRECT
22	TYPE MISMATCH
23	STRING TOO LONG
24	FILE DATA
25	FORMULA TOO COMPLEX
26	CAN'T CONTINUE
27	UNDEF'D FUNCTION
28	VERIFY
29	LOAD

Die Fehlermeldungen 1 bis 9 sind Ein/Ausgabe-bezogene Fehler und werden vom Betriebssystem übermittelt, während die Fehler 10 bis 29 vom BASIC-Interpreter stammen. Wird vom BASIC-

Interpreter ein Fehler erkannt, so wird das X-Register mit der Fehlernummer geladen und nach Adresse $A437 gesprungen, an der der indirekte Sprung JMP ($0300) steht. Wird das Programm jedoch normal mit END beendet, so wird das X-Register zur Unterscheidung von einem Fehler mit einem negativen Wert geladen ($80). Dies wird in der Fehlerroutine geprüft und abhängig davon wird die Fehlerausgabe übergangen und direkt die Meldung 'READY.' ausgegeben und dann in die Eingabewarteschleife verzweigt.

Der Fehlervektor kann von uns zu verschiedenen Zwecken benutzt werden. Zum einen könnten wir die Texte der Fehlermeldungen ändern, z.B. wäre es möglich, deutsche Fehlermeldungen auszugeben. Eine andere, sicher interessantere Möglichkeit besteht darin, beim Auftreten eines Fehlers das Programm nicht abzubrechen, sondern zu einer bestimmten BASIC-Zeile zu verzweigen, wo auf den Fehler entsprechend reagiert werden kann. Eine solche Konstruktion ist meist unter dem Namen

ON ERROR GOTO ...

bekannt und kann z.B. zum Abfangen von Fehlern benutzt werden, die durch Peripheriegeräte verursacht werden.

Die Eingabewarteschleife $302/$303

Wenn der Rechner über den Fehler- und Warmstartvektor $300 eine evtl. Fehlermeldung und sein anschließendes 'READY.' gebracht hat, so springt er über den Vektor $302/$303. Der Rechner wartet dann auf die Eingabe einer Zeile, die durch Return abgeschlossen ist. Dabei wird überwacht, daß die eingegebene Zeile nicht länger als 88 Zeichen, die Länge des BASIC-Eingabepuffers von $200 bis $258 ist. Wird diese Länge

überschritten, so wird die Fehlermeldung 'STRING TOO LONG' ausgegeben. Das erste Zeichen der eingegebenen Zeile entscheidet nun darüber, wie die Zeile weiter verarbeitet wird. War das erste Zeichen eine Ziffer, so geht der Interpreter davon aus, daß wir eine neue BASIC-Zeile eingeben wollen. In diesem Falle wird die komplette Zeilennummer gelesen und nachgesehen, ob diese Zeile schon existiert. Falls ja, so wird die Zeile aus dem Programm gelöscht. Folgt hinter der Zeilennummer nichts mehr, sollte also nur eine Zeile gelöscht werden, so ist die Arbeit schon beendet und es wird wieder an den Beginn der Schleife verzweigt. Folgt noch weiterer Text hinter der Zeilennummer, so wird dieser Text in den Interpreterkode umgewandelt und die Programmzeile in den BASIC-Text eingefügt, ehe auch hier wieder zum Schleifenanfang gesprungen wird.

War das erste eingegebene Zeichen jedoch keine Ziffer, so wird die Zeile als BASIC-Befehl im Direktmodus interpretiert. Die Zeile wird in den Interpreterkode gewandelt, dann wird an die Stelle des Interpreters verzweigt, die einen BASIC-Befehl ausführt.

Auch diesen Vektor können wir für Erweiterungen nutzen. Z.B. wäre es damit möglich, daß die Programmeingabe nicht von der Tastatur erwartet wird, sondern von einem sequentiellen File von Diskette oder vom Userport, an den ein anderer Rechner angeschlossen ist. Dadurch könnte man das Übernehmen von BASIC-Programmen von anderen Rechner entschieden vereinfachen. Das langwierige und fehlerträchtige Abtippen eines Listings entfiele. Bei der direkten Kopplung zweier Rechner brauchte der Senderechner sein Programm lediglich über eine geeignete Schnittstelle zu listen. Ganz besonders würde sich dazu die eingebaute RS-232-Schnittstelle eignen, über die die meisten Rechner verfügen.

Eine weitere Anwendung dieses Vektors ist der AUTO-Befehl. Dieser Befehl erleichtert die Eingabe von Programmen dadurch, daß er nach der Eingabe einer BASIC-Zeile automatisch die nächste Zeilennummer vorgibt und den Cursor dahinter positioniert.

Umwandlung in den Interpreterkode $304/$305

Wie Sie sicher wissen, wird eine Programmzeile nicht so abgespeichert, wie sie eingegeben wird, sondern jedes Befehlswort wird durch einen Ein-Byte-Wert abgekürzt. Das hat gegenüber der direkten Speicherung zwei Vorteile. Zum einen bedeutet es eine Speicherplatzersparnis, wenn statt 5 Bytes für das Wort 'PRINT' nur ein Byte für den Interpreterkode benötigt werden. Der zweite Vorteil macht sich bei der Programmausführung bemerkbar. Wenn der BASIC-Interpreter ein Programm abarbeitet und auf einen Interpreterkode stößt, so kann er daraufhin sofort den zugehörigen Befehl ausführen. Ist der Befehl dagegen im Klartext abgespeichert, so muß das komplette Wort lesen werden. Dann muß der Interpreter seine Befehlstabelle durchsuchen und nachschauen, ob das gelesene Wort als Befehlswort in seiner Tabelle vorhanden ist. Dadurch würde der Programmablauf wesentlich langsamer. Wird die Programmzeile dagegen bei der Eingabe in diese Kodes umgewandelt, so ist diese Wandlung nur einmal erforderlich und nicht bei jeder Ausführung des Befehls. Außerdem ist der Interpreter bei der Programmeingabe sowieso die meiste Zeit mit dem Warten auf unsere Eingabe beschäftigt, so daß die Umwandlung hier am wenigsten stört.

Wollen wir nun neue Befehle nach diesem Verfahren ebenfalls in einen Interpreterkode umwandeln, so können wir diesen Vektor ändern. Unsere Routine muß dann die gelesenen Worte aus der Eingabe mit der Tabelle der neuen Befehlsworte

vergleichen. Wird dabei ein neuer Befehl gefunden, so wird das Befehlswort in der Eingabezeile durch den Interpreterkode ersetzt.

Umwandlung des Interpreterkodes in Klartext $306/$307

Dieser Vektor ist nun für die umgekehrte Aufgabe wie oben zuständig. Wenn wir ein Programm listen wollen, so müssen wir den Interpreterkode wieder zurück in das Befehlswort wandeln. Dabei wird der Befehlskode als Zeiger in die Tabelle der Befehlsworte verwendet. Dieser Vektor betrifft also nur den LIST-Befehl. Wir müssen ihn ändern, wenn wir eigene Interpreterkodes benutzt haben, damit diese richtig aufgelistet werden können. Eine weitere Anwendung wäre eine Modifizierung des LIST-Befehls. Wir könnten dabei zur besseren Lesbarkeit des Listings grundsätzlich nach jedem Befehlswort ein Leerzeichen einfügen oder Schleifenstrukturen durch Einrücken sichtbar machen. Ebenso wäre es möglich, für jedes durch Doppelpunkt getrennte Statement eine eigene Zeile zu verwenden.

Ausführung eines Befehls $308/$309

Dieser Vektor ist einer der wichtigsten überhaupt. Der Vektor zeigt auf die Stelle des Interpreters, die einen BASIC-Befehl ausführt. Normalerweise wird dort ein Zeichen aus dem BASIC-Text geholt und geprüft, ob es sich um den Interpreterkode eines Befehls handelt. Diesen Interpreterkode bezeichnet man oft auch als 'Token'. Handelt es sich um kein Token, so geht der Interpreter davon aus, daß es sich um eine Zuweisung der Form 'A = ...' handelt und verzweigt zum LET-Befehl. Wird ein Token erkannt, so wird das Token als Index in eine Tabelle benutzt, die die Startadressen der BASIC-Befehle enthält. Diese Befehle werden dann als

Subroutine ausgeführt und es kann dann wieder an den Anfang der sogenannten Interpreterschleife zurückgesprungen werden, wo das nächste Statement in gleicher Weise behandelt werden kann.

Mit Hilfe dieses Vektors kann man auf einfache Weise eigene BASIC-Befehle einfügen. Diese kann man durch ein spezielles Zeichen kennzeichnen, z.B. ein Ausrufezeichen '!'. Wir prüfen dann in unserer Routine auf dieses Zeichen und können bei positivem Test den neuen Befehl ausführen.

Haben wir über den oben besprochenen Vektor $304/$305 für unseren neuen Befehl einen eigenen Interpreterkode eingeführt, so ist auch ein spezielles Sonderzeichen nicht mehr erforderlich. Wir prüfen zuerst auf unsere neuen Befehle und verzweigen dann, falls kein neuer Befehl zum Zuge kommt, in die ursprüngliche Routine zur Verarbeitung eines Befehls.

Auswerten eines BASIC-Ausdrucks $30A/$30B

Was der obige Vektor für einen BASIC-Befehl war, ist dieser Vektor für eine Funktion. Dieser Vektor wird benutzt, wenn ein Element eines Ausdruck berechnet werden soll. Dieses Element kann z.B. eine Zahl sein, eine BASIC-Variable oder eine Funktion. Wollen wir neue Funktionen hinzufügen, so müssen wir die Einbindung über diesen Vektor vornehmen. Dabei kann es sich sowohl um numerische als auch um Stringfunktionen handeln. An dieser Stelle müßte man auch einsetzen, wenn man Variablen auf andere Weise abgespeichert hat. Ebenso könnte man eine andere Zahleneingabe realisieren, z.B. mit Hexziffern.

```
PROFI-ASS 64 V2.0    SEITE 1

100:   033C                    .OPT P1,00
110:                   ;
120:                   ; EINGABE VON HEX- UND BINÄRZAHLEN
130:                   ;
140:   030A            AUSDRUCK =   $30A     ; Vektor für Ausdruck auswerten
150:   AE8D            VEKTALT  =   $AE8D    ; alte Routine
160:                   ;
170:   000D            TYP      =   13       ; Variablentyp
180:   0073            CHRGET   =   $73
190:   0079            CHRGOT   =   CHRGET + 6
200:                   ;
210:   BD7E            ADDZIFFER=   $BD7E    ; Einbyteziffer zu FAC addieren
220:                   ;
230:   005D            FLOAT    =   $5D      ; Bereich für Fließkommazahlen
240:   0061            EXP      =   $61      ; Exponent von FAC
250:                   ;
260:   B97E            OVERFLOW =   $B97E    ; OVERFLOW ERROR
270:                   ;
280:   033C                     *=  828
290:                   ;
300:   033C A9 47      INIT     LDA #< TEST
310:   033E A0 03               LDY #> TEST
320:   0340 80 0A 03            STA AUSDRUCK  ; Vektor auf neue Routine setzen
330:   0343 8C 0B 03            STY AUSDRUCK+1
340:   0346 60                  RTS
350:                   ;
360:   0347 A9 00      TEST     LDA #0
370:   0349 85 0D               STA TYP      ; Typflag auf numerisch
380:   034B 20 73 00            JSR CHRGET   ; nächstes Zeichen holen
390:   034E C9 24               CMP #"$"     ; Hexzahl ?
400:   0350 F0 0A               BEQ HEXZAHL
410:   0352 C9 25               CMP #"%"     ; Binärzahl ?
```

```
PROFI-ASS 64 V2.0    SEITE 2

420:   0354 F0 41              BEQ  BINZAHL
430:                   ;
440:   0356 20 79 00           JSR  CHRGOT  ; Flags wieder herstellen
450:   0359 4C 8D AE           JMP  VEKTALT  ; und zur alten Auswertung
460:                   ;
470:   035C 20 80 03 HEXZAHL   JSR  CLRFAC  ; FAC löschen
480:   035F 20 73 00 GETNEXT   JSR  CHRGET  ; nächstes Zeichen holen
490:   0362 90 0B              BCC  ZIFFER  ; Ziffer ?
500:   0364 C9 41              CMP  #"A"
510:   0366 90 1F              BCC  END     ; kleiner als "A" ?
520:   0368 C9 47              CMP  #"F"+1
530:   036A B0 1B              BCS  END     ; größer als "F" ?
540:   036C 38                 SEC
550:   0360 E9 07              SBC  #7      ; Offeset berücksichtigen
560:   036F 38         ZIFFER  SEC
560:   0370 E9 30              SBC  #"0"    ; nach Hex wandeln
570:   0372 48                 PHA          ; Zeichen merken
580:   0373 A5 61              LDA  EXP
580:   0375 F0 07              BEQ  NOCHNULL; ist FAC noch null ?
590:   0377 18                 CLC
600:   0378 69 04              ADC  #4      ; Exponent + 4 => Zahl * 16
610:   037A B0 0E              BCS  OVER    ; Zahl zu groß ?
620:   037C 85 61              STA  EXP
630:   037E 68         NOCHNULL PLA         ; Ziffer wiederholen
640:   037F F0 DE              BEQ  GETNEXT  ; null, dann Addition überflüssig
650:   0381 20 7E BD           JSR  ADDZIFFER  ; Ziffer zu FAC addieren
660:   0384 4C 5F 03           JMP  GETNEXT
670:                   ;
680:   0387 4C 79 00 END       JMP  CHRGOT
690:                   ;
700:   038A 4C 7E B9 OVER      JMP  OVERFLOW
710:                   ;
```

```
PROFI-ASS 64 V2.0    SEITE 3

720:  038D A9 00    CLRFAC  LDA  #0
730:  038F A2 0A            LDX  #10
740:  0391 95 5D    LOOP    STA  FLOAT,X  ; Fließkommabereich löschen
750:  0393 CA               DEX
760:  0394 10 FB            BPL  LOOP
770:  0396 60               RTS
780:                ;
790:  0397 20 8D 03 BINZAHL JSR  CLRFAC   ; FAC löschen
800:  039A 20 73 00 GETBIN  JSR  CHRGET   ; nächstes Zeichen holen
810:  039D C9 32            CMP  #"2"
820:  039F B0 E6            BCS  END      ; größer als "1" ?
830:  03A1 C9 30            CMP  #"0"
840:  03A3 90 E2            BCC  END      ; kleiner als "0" ?
850:  03A5 E9 30            SBC  #"0"     ; von ASCII nach Hex
860:  03A7 48               PHA
870:  03A8 A5 61            LDA  EXP      ; ist Zahl noch null ?
880:  03AA F0 04            BEQ  NULL
890:  03AC E6 61            INC  EXP      ; Zahl verdoppeln
900:  03AE F0 DA            BEQ  OVER     ; zu groß ?
910:  03B0 68       NULL    PLA
920:  03B1 F0 E7            BEQ  GETBIN   ; null nicht addieren
930:  03B3 20 7E BD         JSR  ADDZIFFER  ; Ziffer addieren
940:  03B6 4C 9A 03         JMP  GETBIN   ; und nächste Ziffer holen
]033C-03B9
NO ERRORS
```

Die Routine arbeitet analog wie das Unterprogramm zur Verarbeitung von Dezimalziffern, ist jedoch einfacher und überschaubarer, da keine gebrochenen Zahlen und keine Exponenten berücksichtigt werden müssen. Wenn Sie das Programm mit SYS 828 aktivieren, können Sie ab jetzt

sämtliche Zahleneingaben außer in dezimaler Schreibweise auch hexadezimal oder binär eingeben, z.B.

```
? $FFFF gibt 65535
? %101010 gibt 42
```

Sie sind dabei nicht auf vierstellige Hexzahlen beschränkt, sondern können den vollen Fließkommabereich für Zahlen nutzen. Das bedeutet, daß eine Hexzahl maximal 31 Stellen umfassen kann, z.B.

```
? $FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
```

ergibt

```
2.12676479E+37
```

Bei Binärzahlen können Sie in einer Eingabezeile den vollen Wertebereich gar nicht ausnutzen; eine Zahl mit 78 Binärziffern hat einen Wert von ca. 3E+23.

Mit dieser Befehlserweiterung können Sie Hex- und Binärziffern nicht nur in PRINT-Statements, sondern überall dort verwenden, so sonst Dezimalzahlen vonnöten waren. Besonders im Zusammenhang mit POKE, PEEK und SYS-Befehlen ist die neue Darstellung interessant. So läßt sich die Adresse $D000 für den Videocontroller sicher leichter merken als 53248. Z.B können Sie Sprite 3 nun mit

```
POKE $D015, PEEK($D015) OR %1000
```

aktivieren anstelle von

```
POKE 53248+21, PEEK(53248+21) OR 8
```

Die Hexeingabe hat allerdings noch einen kleinen 'Haken'. Geben Sie einmal

? $ABCDEF

ein. Sie erhalten einen 'SYNTAX ERROR'. Wie kommt es dazu ? Wenn Sie sich die Zahl genau ansehen, so erkennen Sie darin vielleicht das Befehlswort 'DEF' zur Definition von Funktionen. Da der Interpreter unsere Eingabe zuerst in den Interpreterkode wandelt, macht er aus der Folge 'DEF' den zugehörigen Interpreterkode und unsere neue Funktion bringt einen SYNTAX ERROR. Wir können diese Falle jedoch einfach umgehen, indem wir ein Leerzeichen einfügen:

? $ABCD EF

Jetzt erhalten wir den korrekten Wert 11259375. Das Einfügen von Leerzeichen ist möglich, da die CHRGET-Routine Leerzeichen überliest. Dies ist also auch bei normalen Dezimalzahlen der Fall.

Doch schauen wir uns jetzt einmal die Arbeitsweise der Routine an.

Nach der üblichen Initialisierung, die den Vektor auf unsere Routine setzt, wird analog zur Routine des Interpreters das Flag zur Kennzeichnung des Variablentyps gelöscht (auf numerisch gesetzt). Jetzt kann das nächste Zeichen geholt und getestet werden. Ist es ein Dollar- oder Prozentzeichen, womit Hex- bzw. Binärzahlen gekennzeichnet werden, so wird zu unseren neuen Routinen verzweigt. War dies nicht der Fall, werden mit CHRGOT die Flags wieder neu gesetzt und die ursprüngliche Auswertung des Interpreters fortgesetzt. Bei den neuen Routinen gehen wir folgendermaßen vor:

Zu Beginn wird der Fließkommaakku gelöscht, da in ihm unser Ergebnis aufgebaut wird. Nun wird das nächste Zeichen geholt und geprüft, ob es eine Ziffer oder ein Buchstabe von "A" bis "F" ist. Falls diese Bedingung erfüllt ist, wird das Zeichen in den entsprechenden Hexwert umgewandelt; z.B. wird aus "1" der Wert $01 und aus "A" wird $0A. Nun wird der Inhalt des Fließkommaakkus mit 16 multipliziert, falls er nicht gleich null ist. Durch die Multiplikation mit 16 wird der Stellenwert korrekt berücksichtigt. Die Multiplikation lösen wir auf besonders einfache und schnelle Weise. Statt eine Fließkommamultiplikation aufzurufen, die mindestens eine Millisekunde dauern würde, überlegen wir uns, daß eine Multiplikation mit 16 einer Erhöhung des Zweierexponenten um 4 gleichkommt: 16 = 2^4. Wir addieren also einfach 4 zum Exponenten des Fließkommaakkus, was nur ein paar Mikrosekunden dauert. Nachdem wir uns vergewissert haben, daß kein Überlauf vorgekommen ist, holen wir die gerade gelesene Ziffer wieder und addieren sie zum FAC. Handelt es sich bei der gerade gelesenen Zahl um eine Null, dann können wir die Addition übergehen. Dies geschieht in einer Schleife solange, bis die CHRGET-Routine ein Zeichen liest, welches nicht mehr Bestandteil unserer Zahl ist.

Die Umwandlung einer Binärziffer geschieht nach dem gleichen Schema und ist im Grunde noch einfacher. Hier ersetzen wir die Multiplikation mit zwei einfach durch ein Inkrementieren des Zweierexponenten. Das weitere Verfahren stimmt mit der Routine für die Hexzahlen überein.

3.3 Strukturierte Programmierung

Wir haben im Verlaufe des Buches bereits einiges über die Arbeit des BASIC-Interpreters gehört, speziell über die Ausführung einfacher Befehle. Was dabei jedoch noch nicht zur Sprache gekommen ist, ist die Realisierung von Programmstrukturen. Befehle zur strukturierten Programmierung kennt der Interpreter nur zwei:

GOSUB ... RETURN

und

FOR ... NEXT

Will man diese Strukturen verwirklichen, so muß der Interpreter beim Ausführen des RETURN-Befehls wissen, von welcher Stelle aus das Unterprogramm mit GOSUB angesprungen wurde, damit der Rücksprung ordnungsgemäß vonstatten gehen kann. Beim NEXT-Befehl muß zusätzlich zur Adresse des Schleifenbeginns noch der Endwert und die Schrittweite bekannt sein, damit entschieden werden kann, wann die Schleife abzubrechen ist. Man könnte nun diese von RETURN und NEXT gebrauchten Parameter an einer dafür vorgesehenen Stelle im Speicher ablegen. Doch was passiert, wenn man mehrere Unterprogramme oder Schleifen verschachteln will?

Bei jedem RETURN und NEXT muß dafür gesorgt sein, daß die Parameter der jeweils zuletzt angesprochenen Struktur geholt werden können. Was zuletzt gespeichert wurde, muß zuerst wieder geholt werden. Dieses Prinzip ist uns schon vom Stack her bekannt: LAST IN - FIRST OUT. Man benutzt deshalb einfach den Stack, um die Parameter der Programmstrukturen zu speichern.

Welche Daten müssen bei einem GOSUB-Befehl auf dem Stack gespeichert werden? Zuerst einmal sicher die Adresse, von der aus der GOSUB-Aufruf erfolgte. Zum zweiten wird noch die augenblickliche Zeilennummer mit auf dem Stack abgelegt, damit sie nach dem Rücksprung wieder den korrekten Wert hat. Damit man später beim RETURN-Befehl die Daten eines GOSUB-Befehls von denen eines FOR-Befehls unterscheiden kann, legt man zur Identifizierung noch den GOSUB-Kode mit auf den Stack. Ein kompletter Datensatz im Stack sieht dann so aus:

Stapelzeiger →
vor GOSUB-Befehl Programmzeiger hi

Programmzeiger lo

Zeilennummer hi

Zeilennummer lo

GOSUB-Kode $8D

Stapelzeiger
nach GOSUB-Befehl →

Bei einem GOSUB-Befehl wird also Platz für 5 Bytes im Stack benötigt. Da der Stackpointer beim 6510 nur 8 Bit lang ist, kann er nur eine Page adressieren, von $100 bis $1FF. Dadurch wird schon klar, daß nicht beliebig viele Unterprogramme geschachtelt werden können. Maximal wären es 256/5 gleich 51 Unterprogramme. Da der Stack jedoch noch für andere Aufgaben benutzt wird, sind es nicht ganz soviel. Vor der Ausführung eines GOSUB-Befehl wird daher ein Unterprogramm aufgerufen, welches den zur Verfügung stehenden Platz im Stack testet. Diesem Unterprogramm übergibt man im Akku die halbe Anzahl des benötigten Stackspeicherplatzes. Beim GOSUB-Befehl muß

man eine 3 einsetzen, man testet also auf 6 Bytes. Ist der benötigte Platz nicht mehr vorhanden, so wird die Meldung 'OUT OF MEMORY' ausgegeben. Diese Meldung ist leider gleichlautend mit der Meldung, die auftritt, wenn der Speicherplatz für Variable nicht mehr ausreicht. Eine Meldung wie 'STACK OVERFLOW' wäre sicher aussagekräftiger.

Dem BASIC-Interpreter steht im Stack nur der Bereich von $013E bis $1FA zur Verfügung; der Speicherplatz von $0100 bis $0110 dient zur Umwandlung von Fließkommazahlen in Strings und der Platz von $0100 bis $013E wird zur Fehlerkorrektur beim Lesen von der Datasette benutzt.

Was passiert nun beim RETURN-Befehl? Zuerst wird geprüft, ob das oberste Stackelement den Kode für GOSUB enthält. Ist das nicht der Fall, so wird die Fehlermeldung 'RETURN WITHOUT GOSUB' ausgegeben. Ansonsten werden die nächsten vier Bytes vom Stack geholt und damit die Parameter für Zeilennummer und Programmzeiger versorgt. Der Stackpointer steht nun wieder auf dem Wert, den er vor dem Aufruf von GOSUB hatte. Wenn jetzt zur Interpreterschleife gesprungen wird, wird die Programmausführung automatisch nach dem GOSUB-Befehl fortgesetzt.

Bei einer FOR-NEXT-Schleife ist das Prinzip ähnlich; aufgrund von mehreren Parametern, die zwischengespeichert werden müssen, jedoch etwas komplizierter. Die benötigten Parameter werden in der folgenden Reihenfolge auf dem Stack abgelegt:

Stapelzeiger →
vor FOR-Befehl Programmzeiger hi

Programmzeiger lo

Zeilennummer hi

Zeilennummer lo

Mantisse 4 }
Mantisse 3 }
Mantisse 2 } TO-Wert
Mantisse 1 }
Exponent }

Vorzeichen }
Mantisse 4 }
Mantisse 3 }
Mantisse 2 } STEP-Wert
Mantisse 1 }
Exponent }

Variablenadresse hi

Variablenadresse lo

FOR-Kode $81

Stapelzeiger
nach FOR-Befehl →

Sie sehen also, daß eine FOR-NEXT-Schleife 18 Bytes Speicherplatz im Stack benötigt. Bei einem NEXT-Befehl passiert nun folgendes. Zuerst wird geprüft, ob das oberste Stackelement den FOR-Kode $81 enthält. Ist das nicht der Fall, wird die Fehlermeldung 'NEXT WITHOUT FOR' ausgegeben. Folgt nach dem NEXT-Befehl noch ein Variablenname, so wird die Adresse der Variablen ermittelt und mit der Variablenadresse auf dem Stack verglichen. Bei Gleichheit oder falls kein Variablenname angegeben wurde, wird der Variablenwert in den FAC geholt und der STEP-Wert vom Stack dazu addiert. Dieser Wert wird nun als neuer Variablenwert abgespeichert und kann mit dem auf dem Stack stehenden Endwert verglichen werden. Abhängig vom Vorzeichen des STEP-Werts kann nun entschieden werden, ob die Schleife beendet ist oder nicht. Kann der Schleifendurchlauf beendet werden, so wird der Stapelzeiger um 18 erhöht, um den Datensatz aus dem Stack zu entfernen, und zur Interpreterschleife gesprungen, wo das nächste Statement ausgeführt werden kann. War der Endwert dagegen noch nicht erreicht, werden Zeilennummer und Programmzeiger vom Stack geladen. Dabei bleibt der Stackpointer jedoch unverändert, damit der Datensatz beim nächsten NEXT-Befehl noch vorhanden ist.

Folgte nach dem NEXT-Befehl noch ein Variablenname, deren Adresse nicht mit der auf dem Stack gespeicherten übereinstimmte, so wird der Stackpointer ebenfalls um 18 erhöht und geprüft, ob noch ein FOR-NEXT-Datensatz auf dem Stack ist. Dadurch wird automatisch eine eventuell vorhandene innere Schleife geschlossen.

Mit diesen Kenntnissen können wir versuchen, eine neue Struktur zu schaffen. Falls Sie schon einmal in PASCAL programmiert haben, kennen Sie sicher die REPEAT...UNTIL-Schleife. Dies ist eine Programmstruktur, die solange durchlaufen wird,

bis ein Abbruchkriterium erfüllt ist, z.B.

```
REPEAT
I=I+1
UNTIL I=10
```

Hier wird die Schleife solange durchlaufen, bis die Endbedingung I=10 erfüllt ist. Diese Struktur läßt sich sehr vielseitig einsetzen. Wie die FOR-NEXT-Schleife wird sie mindestens einmal durchlaufen. Auch das Warten auf einen Tastendruck kann man mit dieser Schleife realisieren.

```
REPEAT : GET A$ : UNTIL A$<>""
```

oder noch einfacher

```
REPEAT : UNTIL PEEK(197)<>64
```

Hier wird solange gewartet, bis die Speicherzelle 197 einen Wert von ungleich 64 hat, was gleichbedeutend mit dem Drücken einer Taste ist.

Das folgende Maschinenprogramm implementiert diese Struktur im BASIC.

```
REPEAT - UNTIL     PROFI-ASS 64 V2.0    SEITE 1

100:   033C                  .OPT P,00
110:   033C                  .TIT "REPEAT - UNTIL"
110:                  ;
120:                  ; REPEAT - UNTIL - SCHLEIFE
130:                  ;
140:   0308           BEFEHL  =   $308    ; Vektor für Befehl ausführen
```

```
REPEAT - UNTIL     PROFI-ASS 64 V2.0    SEITE 2

150:   A7E7             BEF.ALT  =    $A7E7    ; alte Routine
160:                    ;
170:   0022             ADR      =    $22      ; Adresse für Fehlermeldung
180:   0039             LINENO   =    $39      ; aktuelle Zeilennummer
190:   0073             CHRGET   =    $73
200:   0079             CHRGOT   =    CHRGET + 6
210:   007A             TXTPTR   =    CHRGOT + 1
220:   0100             STACK    =    $100     ; Prozessorstack
230:   A445             ERROR    =    $A445    ; Fehlermeldung ausgeben
240:                    ;
250:   A3FB             TESTSTACK=    $A3FB    ; auf Platz im Stack testen
260:   AD8A             FRMNUM   =    $AD8A    ; numerischen Ausdruck holen
270:   A7AE             INTER    =    $A7AE    ; Interpreterschleife
280:   AF08             SYNTAX   =    $AF08    ; SYNTAX ERROR
290:   A906             NEXTSTAT =    $A906    ; nächstes Statement suchen
300:                    ;
310:   033C                      *=   828
320:   033C A9 47       INIT     LDA  #< TEST
330:   033E A0 03                LDY  #> TEST
340:   0340 8D 08 03             STA  BEFEHL   ; Vektor auf neue Routine
350:   0343 8C 09 03             STY  BEFEHL+1
360:   0346 60                   RTS
370:                    ;
380:   0347 20 73 00 TEST        JSR  CHRGET   ; nächstes Zeichen holen
390:   034A C9 21                CMP  #"!"
400:   034C F0 06                BEQ  NEWBEF   ; neuer Befehl ?
410:                    ;
420:   034E 20 79 00             JSR  CHRGOT   ; Flags wieder herstellen
430:   0351 4C E7 A7             JMP  BEF.ALT   ; und alte Befehle ausführen
440:                    ;
450:   0354 20 73 00 NEWBEF      JSR  CHRGET   ; nächstes Zeichen
460:   0357 C9 52                CMP  #"R"     ; Repeat-Befehl ?
```

```
470:   0359 F0 07              BEQ  REPEAT
480:   035B C9 55              CMP  #"U"     ; Until-Befehl ?
490:   035D F0 24              BEQ  UNTIL
500:   035F 4C 08 AF SYNERR    JMP  SYNTAX   ; sonst SYNTAX ERROR
510:                  ;
520:   0362 20 73 00 REPEAT    JSR  CHRGET   ; Zeiger auf nächstes Zeichen
530:   0365 A9 03              LDA  #3
540:   0367 20 FB A3           JSR  TESTSTACK  ; genügend Platz im Stack ?
550:   036A 20 06 A9           JSR  NEXTSTAT  ; nächstes Statement suchen
560:   0360 18                 CLC
570:   036E 98                 TYA           ; Offset auf nächsten Befehl
580:   036F 65 7A              ADC  TXTPTR   ; addieren
590:   0371 48                 PHA           ; und auf Stack
600:   0372 A5 7B              LDA  TXTPTR+1
610:   0374 69 00              ADC  #0
620:   0376 48                 PHA
630:   0377 A5 39              LDA  LINENO   ; Zeilennummer
640:   0379 48                 PHA           ; ebenfalls auf Stack
650:   037A A5 3A              LDA  LINENO+1
660:   037C 48                 PHA
670:   037D A9 52              LDA  #"R"     ; und REPEAT-Kode
680:   037F 48                 PHA           ; auf Stack
690:   0380 4C AE A7           JMP  INTER    ; zur Interpreterschleife
700:                  ;
710:   0383 20 73 00 UNTIL     JSR  CHRGET   ; folgt Bedingung ?
720:   0386 F0 D7              BEQ  SYNERR   ; nein, dann Fehler
730:   0388 20 8A AD           JSR  FRMNUM   ; Bedingung auswerten
740:   038B A8                 TAY           ; Ergebnis merken
750:   038C BA                 TSX           ; Stackpointer nach X
760:   038D BD 01 01           LDA  STACK+1,X  ; letzten Stack-Eintrag
770:   0390 C9 52              CMP  #"R"     ; auf REPEAT-Kode testen
780:   0392 D0 23              BNE  RPTERR   ; nicht, dann Fehlermeldung
```

```
790:   0394 98                 TYA
800:   0395 D0 17              BNE  RPTENDE ; Ausdruck wahr, Schleife beenden
800:                  ;
810:   0397 BD 02 01           LDA  STACK+2,X
820:   039A 85 3A              STA  LINENO+1  ; Zeilenummer
830:   039C BD 03 01           LDA  STACK+3,X
840:   039F 85 39              STA  LINENO
850:   03A1 BD 04 01           LDA  STACK+4,X
860:   03A4 85 7B              STA  TXTPTR+1  ; und Programmzeiger
870:   03A6 BD 05 01           LDA  STACK+5,X ; vom Stack holen
880:   03A9 85 7A              STA  TXTPTR
890:   03AB 4C AE A7           JMP  INTER    ; zur Interpreterschleife
900:                  ;
910:   03AE 8A        RPTENDE  TXA           ; Stackpointer
920:   03AF 18                 CLC
930:   03B0 69 05              ADC  #5       ; um 5 erhöhen
940:   03B2 AA                 TAX
950:   03B3 9A                 TXS
960:   03B4 4C AE A7           JMP  INTER    ; und zur Interpreterschleife
970:                  ;
980:   03B7 A9 C0     RPTERR   LDA  #< TEXT
990:   03B9 85 22              STA  ADR      ; Zeiger auf Fehlermeldung setzen
1000:  03BB A9 03              LDA  #> TEXT
1010:  03BD 4C 45 A4           JMP  ERROR
1020:                 ;
1030:  03C0 55 4E 54 TEXT     .ASC "UNTIL WITHOUT REPEAT"
]033C-03D4
NO ERRORS
```

Doch nun zur Benutzung unserer neuen Befehlserweiterung. Wir kennzeichnen der Einfachheit halber unsere neuen Befehle durch ein vorangestelltes Ausrufezeichen '!' und ein 'R' für REPEAT bzw. ein 'U' für UNTIL. Wenn Sie das Maschinenprogramm assembliert und mit SYS 828 aktiviert haben, so probieren Sie es doch gleich mal aus:

```
100 I=0
110 !R
120 I=I+1 : PRINT I
130 !U I=10
```

Das Programm liefert die Zahlen von 1 bis 10. Auch geschachtelte Anweisungen sind möglich.

```
100 I=0
110 !R
120 I=I+1 : PRINT "I=" ; I : J=0
130 !R
140 J=J+1 : PRINT "J=" ; J
150 !U J=3
160 !U I=3
```

In diesen geschachtelten Schleifen läuft der Zähler I von 1 bis 3 und bei jedem I die innere Schleife J von 1 bis 3. Das obige Problem ließe sich sicher mit zwei FOR-NEXT-Schleifen einfacher lösen. Das Hauptanwendungsgebiet der REPEAT-UNTIL-Schleife sind Strukturen, bei denen die Anzahl der Durchläufe vor Beginn noch nicht festliegt, sondern sich erst im Laufe des Programms ergibt. Das Abbruchkriterium kann z.B. eine gedrückte Taste sein. Sehr nützlich ist diese Programmstruktur auch bei Iterationen, z.B. die Berechnung der Quadratwurzel nach Newton.

```
100 INPUT "EINGABE "; A
110 X1 = A
120 !R
130 X0 = X1
140 X1 = (X0 + A/X0)/2
150 !U ABS (X1-X0) < 1E-8
160 PRINT "DIE WURZEL IST "; X1
```

Hier wird solange ein Näherungswert berechnet, bis die Abweichung zweier aufeinander folgender Werte kleiner als 10^-8 ist. Probieren Sie einmal ein paar Werte und vergleichen Sie das Ergebnis mit der SQR-Funktion.

Mit dieser Struktur kann man auch eine Endlosschleife programmieren, indem man als Abruchkriterium eine Bedingung nimmt, die nie wahr ist, z.B.

```
100 !R
110 PRINT TI
120 !U 1=0
```

Diese Schleife wird vom Programm her nie abgebrochen.

Die REPEAT-UNTIL-Schleife arbeitet schneller als eine IF ... GOTO Abfrage, da bei GOTO jedesmal die Zeile, zu der gesprungen wird, gesucht werden muß. Beim UNTIL-Befehl braucht die Adresse nur vom Stack geholt werden. Außerdem wird das Programm dadurch übersichtlicher, da die Absichten des Programmierers deutlicher zum Ausdruck kommen.

Kommen wir jetzt zur Besprechung des Maschinenprogramms. Wir gehen darin ähnlich vor wie bei den schon vorhandenen Programmstrukturen. Nach der üblichen Initialisierung, in der der Vektor zur Befehlsauswertung auf unsere Routine gesetzt

wird, testet das Programm zuerst, ob wie einen neuen Befehl verwenden wollen. Falls kein Ausrufezeichen gefunden wurde, wird zur alten Befehlsauswertung zurückgekehrt. Dann wird das nächste Zeichen geholt und auf "R" und "U" geprüft. Abhängig davon wird in die Routinen REPEAT und UNTIL verzweigt. Haben wir dagegen keins dieser Zeichen gelesen, so springen wir zur Fehlermeldung 'SYNTAX ERROR'.

Beim REPEAT-Befehl setzen wir den Programmzeiger durch Aufruf von CHRGET auf das nächste Zeichen und prüfen dann, ob noch genügend Platz auf dem Stack ist. Mit der Routine NEXTSTAT suchen wir den nächsten Befehl, dessen relative Adresse wir im Y-Register zurückbekommen. Dieses Wert addieren wir zum Programmzähler und legen ihn auf dem Stack ab. Anschließend wird die Zeilennummer ebenfalls auf den Stack gelegt. Zur Kennzeichnung des Datensatzes als REPEAT-Befehl legen wir noch den Buchstaben "R" auf den Stack. Der Datensatz im Stack ist also analog wie beim GOSUB-Befehl aufgebaut. Damit ist die Arbeit schon beendet und wir kehren zur Interpreterschleife zurück.

Der UNTIL-Befehl prüft, ob eine Bedingung folgt und wertet diese aus. Das Ergebnis wird im Y-Register gespeichert. Jetzt laden wir das X-Register mit dem Stackpointer und vergleichen das oberste Stackelement mit "R", dem Kennzeichen für REPEAT. Haben wir keine Übereinstimmung festgestellt, so geben wir die Fehlermeldung 'UNTIL WITHOUT REPEAT' aus. Beachten Sie dabei, daß das letzte Zeichen der Fehlermeldung geshiftet eingegeben werden muß (gesetztes Bit 7). Haben wir dagegen das "R" gefunden, so ist das weitere Vorgehen abhängig vom Wert der Bedingung. War die Bedingung nicht erfüllt, so laden wir Zeilennummer und Programmzeiger vom Stack und springen zur Interpreterschleife. Beachten Sie, daß wir die Daten nicht mit PLA vom Stack holen, sondern mit LDA STACK,X ,

nachdem vorher der Stackpointer ins X-Register kopiert wurde. Dadurch bleibt der Wert des Stackpointers erhalten und die Daten stehen uns beim nächsten UNTIL-Befehl weiterhin zur Verfügung. War die Bedingung jedoch erfüllt, so erhöhen wir einfach den Stackpointer um 5. Damit ist der Datensatz aus dem Stack entfernt, und wir fahren mit dem nächsten Befehl fort.

3.4 Die Verwendung neuer Schlüsselworte

Die komfortabelste Art und Weise, neue Befehle einzufügen, besteht natürlich darin, ihnen einen Namen zu geben, über den sie dann aufgerufen werden können. Intern wird dieses Schlüsselwort in Form eines Tokens gespeichert, das ist der Interpreterkode, der Werte von $80 bis $FF haben kann.

Beim Commodore 64 sind die Token von $80 bis $CB benutzt sowie $FF als Kode für Pi. Wenn wir also neue Schlüsselworte einfügen wollen, so stehen uns dafür noch die Interpreterkodes $CC (204) bis $FE (254) zur Verfügung. Wir könnten also bis zu 51 neue Befehle einfügen. Überlegen wir einmal, was dazu alles notwendig wäre.

Zuerst muß eine Routine existieren, die bei der Eingabe einer BASIC-Zeile die neuen Schlüsselworte in Token umwandelt. Dann muß die Routine zur Ausführung der Befehle die neuen Token erkennen und das zugehörige Programm zur Ausführung dieses Befehls aufrufen. Damit wir unser Programm listen können, muß auch die List-Routine dermaßen modifiziert werden, daß sie zu den neuen Token die entsprechenden Befehlsworte im Klartext ausgibt. Dazu legen wir am günstigsten unsere neuen Befehlsworte und die Adressen der zugehörigen Routinen in je einer Tabelle ab, genau so wie dies der Interpreter mit den Standardbefehlen handhabt.

Erinnern wir uns an die BASIC-Vektoren, so finden wir vier Vektoren, die für diese Aufgaben zuständig sind. Die Vektoren für BASIC-Befehl ausführen ($308) und Funktion berechnen ($30A) haben wir dabei schon benutzt. Für die Umwandlung in Tokens benötigen wir nun noch die Vektoren $304 sowie $306, der die neuen Tokens korrekt listet.

Haben wir diese Routinen einmal erstellt, so wird es dann ein leichtes sein, neue Schlüsselworte hinzuzufügen. Wir brauchen dazu nur die Befehlsworte sowie die Adressen, an denen die zugehörigen Routinen stehen, in die Tabellen dieses Programms einzutragen.

Dieses Verfahren ist auch schneller in der Ausführung, da nicht spezielle Sonderzeichen wie '!' zur Erkennung der neuen Befehle einfügt werden müssen. Im Programm selbst sieht der Befehl 'REPEAT' sicher besser aus als '!R'.

Ehe wir uns nun an eine Routine wagen, die neue Schlüsselworte in Tokens umwandelt, schauen wir uns erst einmal an, wie der Interpreter diese Sache handhabt. Wir haben dazu die ROM-Routine für Sie neu assembliert. Wenn wir das Prinzip durchschaut haben, sollte es uns dann nicht schwer fallen, diese Routine so zu modifizieren, daß wir eigene Tokens erzeugen können.

```
PROFI-ASS 64 V2.0    SEITE 1

100:  A57C                 .OPT P
110:              ;
120:              ; ROM-ROUTINE ZUR UMWANDLUNG IN TOKENS
130:              ;
140:              ; Besondere Tokens
150:              ;
160:  0083        DATA   =   $83
170:  008F        REM    =   $8F
180:  0099        PRINT  =   $99
190:              ;
200:  0008        CHAR   =   8       ; aktuelles Zeichen
210:  000B        COUNT  =   11      ; Zähler für Befehlsworte
220:  0071        PNT    =   $71     ; Zeiger in umgewandelte Zeile
```

```
0022              QUOTE    =    $22       ; Hochkomma
000F              FLAG     =    15        ; Flag bei DATA und REM
007A              TXTPTR   =    $7A       ; Zeiger in umzuwandelnde Zeile
0200              BUFFER   =    $200      ; Eingabepuffer
                  ;
A09E              TABLE    =    $A09E     ; Tabelle der Befehlsworte
                  ;
A57C                       *=   $A57C     ; ROM-Routine
                  ;
A57C A6 7A                 LDX  TXTPTR    ; Zeiger auf erstes Zeichen
A57E A0 04                 LDY  #4        ; Zeiger in umgewandelte Zeile
A580 84 0F                 STY  FLAG       ; Flag löschen
A582 BD 00 02 NEXTCHAR LDA  BUFFER,X   ; Zeichen aus Puffer holen
A585 10 07                 BPL  NORMAL
A587 C9 FF                 CMP  #$FF      ; Kode für 'Pi' ?
A589 F0 3E                 BEQ  TAKCHAR   ; ja, Kode so übernehmen
A58B E8                    INX            ; sonst Zeichen ignorieren
A58C D0 F4                 BNE  NEXTCHAR
                  ;
A58E C9 20        NORMAL   CMP  #" "      ; Leerzeichen ?
A590 F0 37                 BEQ  TAKCHAR   ; so übernehmen
A592 85 08                 STA  CHAR      ; Zeichen merken
A594 C9 22                 CMP  #QUOTE    ; Hochkomma ?
A596 F0 56                 BEQ  GETCHAR ; ja
A598 24 0F                 BIT  FLAG      ; Flag testen
A59A 70 20                 BVS  TAKCHAR   ; DATA-Modus, so übernehmen
A59C C9 3F                 CMP  #"?"      ; Fragezeichen
A59E D0 04                 BNE  SKIP
A5A0 A9 99                 LDA  #PRINT    ; durch PRINT-Kode ersetzen
A5A2 D0 25                 BNE  TAKCHAR
A5A4 C9 30        SKIP     CMP  #"0"      ; kleiner als '0' ?
A5A6 90 04                 BCC  SKIP1
```

```
PROFI-ASS 64 V2.0    SEITE 3

550:   A5A8 C9 3C              CMP  #"<"      ; kleiner als '<' ?
560:   A5AA 90 1D              BCC  TAKCHAR   ; ja, Zeichen so übernehmen
570:   A5AC 84 71     SKIP1    STY  PNT       ; Zeiger in Zeile merken
580:   A5AE A0 00              LDY  #0
590:   A5B0 84 0B              STY  COUNT     ; Zähler für Befehlsworte auf null
600:   A5B2 88                 DEY
610:   A5B3 86 7A              STX  TXTPTR    ; Zeiger in Zeile merken
620:   A5B5 CA                 DEX
630:                  ;
640:   A5B6 C8        CMPLOOP  INY            ; Zeiger in Befehlstabelle
640:   A5B7 E8                 INX            ; und Zeiger in Zeile erhöhen
650:   A5B8 BD 00 02 TESTNEXT LDA  BUFFER,X   ; Zeichen aus Puffer holen
660:   A5BB 38                 SEC
670:   A5BC F9 9E A0           SBC  TABLE,Y   ; mit Befehlswort vergleichen
680:   A5BF F0 F5              BEQ  CMPLOOP   ; gleich, dann nächstes Zeichen
690:   A5C1 C9 80              CMP  #$80      ; letzter Buchstabe ?
700:   A5C3 D0 30              BNE  NEXTCMD   ; sonst Zeiger auf nächsten Befehl
710:   A5C5 05 0B              ORA  COUNT     ; gefunden, Nr+$80 = Interpreterkode
720:   A5C7 A4 71     TAKCHAR1 LDY  PNT       ; Zeiger zurückholen
730:                  ;
740:   A5C9 E8        TAKCHAR  INX
740:   A5CA C8                 INY
750:   A5CB 99 FB 01           STA  BUFFER-5,Y  ; Kode abspeichern
760:   A5CE B9 FB 01           LDA  BUFFER-5,Y  ; Flags wiederherstellen
770:   A5D1 F0 36              BEQ  ENDE      ; Zeilenende ?
780:   A5D3 38                 SEC
790:   A5D4 E9 3A              SBC  #":"      ; Trennzeichen ?
800:   A5D6 F0 04              BEQ  SKIP2     ; DATA-Flag löschen
810:   A5D8 C9 49              CMP  #0ATA-":"  ; Kode für 'DATA'
820:   A5DA D0 02              BNE  SKIP3
830:   A5DC 85 0F     SKIP2    STA  FLAG       ; bei 'DATA' Bit 6 setzen
840:   A5DE 38        SKIP3    SEC
```

```
850:   A5DF E9 55              SBC  #REM-":"   ; Kode für 'REM'
860:   A5E1 D0 9F              BNE  NEXTCHAR   ; nein, nächstes Zeichen holen
870:   A5E3 85 08              STA  CHAR     ; Nullbyte bei 'REM' abspeichern
880:   A5E5 BD 00 02 REMLOOP   LDA  BUFFER,X
890:   A5E8 F0 DF              BEQ  TAKCHAR  ; Zeilenende, Zeichen so übernehmen
900:   A5EA C5 08              CMP  CHAR     ; nächstes '"' oder REM oder DATA
910:   A5EC F0 DB              BEQ  TAKCHAR  ; ja ?
920:   A5EE C8         GETCHAR INY
930:   A5EF 99 FB 01           STA  BUFFER-5,Y  ; Zeichen übernehmen
940:   A5F2 E8                 INX
950:   A5F3 D0 F0              BNE  REMLOOP
960:                   ;
970:   A5F5 A6 7A      NEXTCMD LDX  TXTPTR   ; Zeilenzeiger auf Anfang des Worts
980:   A5F7 E6 0B              INC  COUNT    ; Zähler auf nächstes Befehlswort
990:   A5F9 C8         WEITER  INY
1000:  A5FA B9 9D A0           LDA  TABLE-1,Y  ; nächster Buchstabe
1010:  A5FD 10 FA              BPL  WEITER   ; Wort noch nicht zuende ?
1020:  A5FF B9 9E A0           LDA  TABLE,Y
1030:  A602 D0 B4              BNE  TESTNEXT   ; auf nächstes Befehlswort testen
1040:                  ;
1050:  A604 BD 00 02           LDA  BUFFER,X
1060:  A607 10 BE              BPL  TAKCHAR1   ; Zeichen so übernehmen
1070:                  ;
1080:  A609 99 FD 01 ENDE      STA  BUFFER-3,Y ; Puffer mit null abschließen
1090:                  ;
1100:  A60C C6 7B              DEC  TXTPTR+1
1110:  A60E A9 FF              LDA  #$FF     ; TXTPTR auf $01FF, BUFFER-1
1120:  A610 85 7A              STA  TXTPTR
1130:  A612 60                 RTS
]A57C-A613
NO ERRORS
```

Wenn eine Zeile in Interpreterkode umgewandelt werden soll, so muß sie im BASIC-Eingabepuffer von $200 bis $258 stehen. Der Zeiger TXTPTR ($7A/$7B) muß dabei auf das erste Zeichen in der Zeile hinter der Zeilenummer zeigen. Zu Beginn wird das X-Register mit diesem Zeiger geladen. Das X-Register dient nun in der ganzen Routine als Zeiger in die noch nicht umgewandelte Zeile, während das Y-Register der Zeiger in die umgewandelte Zeile ist. Nachdem das FLAG gelöscht wurde, wird das erste Zeichen der Zeile geladen. Ist der Kode dieses Zeichens größer als $7F, wird es auf den Kode 255 für Pi überprüft. Falls der Test positiv ausfällt, wird das Zeichen so übernommen. Alle anderen Zeichen mit gesetztem Bit 7 werden ignoriert; der Zeiger wird erhöht und das nächste Zeichen wird getestet. War das Zeichen jedoch ein normales 'ungeshiftetes' Zeichen, so wird jetzt auf spezielle Zeichen geprüft. Ein Leerzeichen wird unverändert übernommen. Ansonsten wird das augenblickliche Zeichen in CHAR gespeichert. Handelte es sich um ein Hochkomma, so wird nach GETCHAR verzweigt, wo die Zeichen solange unverändert übernommen werden, bis ein weiteres Hochkomma gefunden wird. Durch Testen von FLAG wird geprüft, ob ein DATA-Befehl erkannt wurde. In diesem Fall wird der nachfolgende Text unverändert übernommen. Als nächstes wird der Kode für '?' gegen das Token von 'PRINT' ersetzt. Nachdem die Ziffern und die Zeichen ':' und ';' herausgefiltert wurden, die unverändert übernommen werden, kommt nun die eigentliche Umwandlung in Tokens.

Der Zeiger in die umgewandelte Zeile wird in PNT gespeichert, der Zähler für die Nummer des Befehlswortes wird initialisiert. Ab dem Label CMPLOOP wird nun der Vergleich durchgeführt. Vom augenblicklichen Zeichen im Puffer wird der erste Buchstabe aus der Befehlworttabelle abgezogen. Waren die Zeichen gleich, so wird das nächste Zeichen mit dem

zweiten Buchstaben verglichen. Wurde Ungleichheit festgestellt, so wird die Differenz auf $80 geprüft. Dieser Wert ergibt sich dann, wenn das letzte Zeichen eines Befehls in der Befehlsworttabelle erreicht wurde, da es mit gesetztem Bit 7 abgespeichert wurde. War das der Fall, so enthält der Akku noch die Differenz $80. Durch logisches Odern mit der Befehlsnummer COUNT erhält man den Interpreterkode, der jetzt abgespeichert wird. Wurde jedoch keine Übereinstimmung mit der Befehlstabelle festgestellt, so wird bei NEXTCMD der Anfang des nächsten Befehlsworts gesucht und der Zähler für die Nummer des Befehlsworts um eins erhöht. Falls die Tabelle noch nicht zu Ende ist, wird zur Vergleichsschleife zurückgesprungen und mit dem nächsten Wort aus der Tabelle verglichen. War das Ende der Tabelle erkannt (durch ein Nullbyte gekennzeichnet), so wird das momentane Zeichen unverändert übernommen.

Nachdem ab dem Label TAKCHAR der Interpreterkode oder das gelesene Zeichen abgespeichert wurde, werden nun spezielle Zeichen behandelt. Wird der Doppelpunkt erkannt, so wird FLAG gelöscht und damit ein evtl. DATA-Modus aufgehoben, der sonst bei einem DATA-Kode gesetzt wird. Wird der REM-Befehl erkannt, wird als augenblickliches Zeichen eine Null abgespeichert, was in der nächsten Schleife REMLOOP dazu führt, daß alle Zeichen bis zur Null (Zeilenende) unverändert übernommen werden. Zum Ende der kompletten Routine (Label ENDE) wird der umgewandelte Puffer mit Null abgeschlossen und der TXTPTR auf ein Zeichen vor den Eingabepuffer gesetzt.

Wenn wir jetzt selbst Befehlsworte in Tokens umwandeln wollen, so müssen wir dafür sorgen, daß nach dem Durchsuchen der Befehlstabelle im ROM noch die Tabelle mit unseren eigenen Befehlsworten durchsucht wird. Desweiteren muß noch festgelegt werden, welche Tokens wir für die neuen Befehle

vergeben wollen. Anbieten würden sich die Tokens ab $CC, direkt anschließend an die bereits vorhandenen Tokens.

```
PROFI-ASS 64 V2.0    SEITE 1

100:  C000                    .OPT P,00
110:                 ;
120:                 ; ROUTINE ZUR BENUTZUNG EIGENER TOKENS
130:                 ;
140:                 ; Besondere Tokens
150:                 ;
160:  0083           DATA     =    $83
170:  008F           REM      =    $8F
180:  0099           PRINT    =    $99
190:                 ;
200:  0008           CHAR     =    8
210:  000B           COUNT    =    11
220:  0071           PNT      =    $71
230:  0022           QUOTE    =    $22      ; Hochkomma
240:  000F           FLAG     =    15       ;
250:  007A           TXTPTR   =    $7A
260:  0200           BUFFER   =    $200     ; Eingabepuffer
270:                 ;
280:  A09E           TABLE    =    $A09E    ; Tabelle der Befehlsworte
290:                 ;
300:  C000                    *=   $C000    ; neue Routine
310:                 ;
320:  C000 A6 7A              LDX  TXTPTR   ; Zeiger auf erstes Zeichen
330:  C002 A0 04              LDY  #4       ; Zeiger in umgewandelte Zeile
340:  C004 84 0F              STY  FLAG      ; Flag für spezielle Zeichen
350:  C006 BD 00 02 NEXTCHAR LDA  BUFFER,X  ; Zeichen aus Puffer holen
360:  C009 10 07              BPL  NORMAL
370:  C00B C9 FF              CMP  #$FF     ; Kode für 'Pi' ?
380:  C000 F0 3E              BEQ  TAKCHAR  ; ja, Kode so übernehmen
```

```
PROFI-ASS 64 V2.0    SEITE 2

390:   C00F E8                 INX               ; sonst Zeichen ignorieren
400:   C010 D0 F4              BNE  NEXTCHAR
410:                   ;
420:   C012 C9 20      NORMAL  CMP  #" "         ; Leerzeichen ?
430:   C014 F0 37              BEQ  TAKCHAR   ; so übernehmen
440:   C016 85 08              STA  CHAR      ; Zeichen merken
450:   C018 C9 22              CMP  #QUOTE    ; Hochkomma ?
460:   C01A F0 55              BEQ  GETCHAR
470:   C01C 24 0F              BIT  FLAG        ; DATA-Modus ?
480:   C01E 70 20              BVS  TAKCHAR   ; ja, so übernehmen
490:   C020 C9 3F              CMP  #"?"      ; Fragezeichen ?
500:   C022 D0 04              BNE  SKIP
510:   C024 A9 99              LDA  #PRINT    ; durch PRINT-Kode ersetzen
520:   C026 D0 25              BNE  TAKCHAR
530:   C028 C9 30      SKIP    CMP  #"0"      ; kleiner als '0' ?
540:   C02A 90 04              BCC  SKIP1
550:   C02C C9 3C              CMP  #"<"      ; kleiner als '<' ?
560:   C02E 90 1D              BCC  TAKCHAR   ; ja, Zeichen so übernehmen
570:   C030 84 71      SKIP1   STY  PNT       ; Zeiger in Zeile merken
580:   C032 A0 00              LDY  #0
590:   C034 84 0B              STY  COUNT     ; Zähler für Befehlsworte auf Null
600:   C036 88                 DEY
610:   C037 86 7A              STX  TXTPTR
620:   C039 CA                 DEX
630:                   ;
640:   C03A C8         CMPLOOP INY
640:   C03B E8                 INX            ; Zeiger erhöhen
650:   C03C BD 00 02 TESTNEXT LDA  BUFFER,X  ; Zeichen aus Puffer holen
660:   C03F 38                 SEC
670:   C040 F9 9E A0           SBC  TABLE,Y   ; mit Befehlsworten vergleichen
680:   C043 F0 F5              BEQ  CMPLOOP   ; gleich, dann nächstes Zeichen
690:   C045 C9 80              CMP  #$80      ; letzter Buchstabe ?
```

```
700:   C047 D0 2F              BNE  NEXTCMD   ; sonst Zeiger auf nächsten Befehl
710:   C049 05 0B              ORA  COUNT    ; Nr+$80 = Interpretercode
720:   C04B A4 71     TAKCHAR1 LDY  PNT      ; Zeiger zurückholen
730:                  ;
740:   C04D E8        TAKCHAR  INX
740:   C04E C8                 INY
750:   C04F 99 FB 01           STA  BUFFER-5,Y  ; Kode abspeichern
760:   C052 C9 00              CMP  #0       ; Flags wiederherstellen
770:   C054 F0 38              BEQ  ENDE     ; Zeilenende ?
780:   C056 38                 SEC
790:   C057 E9 3A              SBC  #":"     ; Trennzeichen ?
800:   C059 F0 04              BEQ  SKIP2
810:   C05B C9 49              CMP  #DATA-":"  ; Kode für 'DATA' ?
820:   C05D D0 02              BNE  SKIP3
830:   C05F 85 0F     SKIP2    STA  FLAG      ; bei 'DATA' bit 6 setzen
840:   C061 38        SKIP3    SEC
850:   C062 E9 55              SBC  #REM-":"  ; Kode für 'REM' ?
860:   C064 D0 A0              BNE  NEXTCHAR  ; nein, nächstes Zeichen holen
870:   C066 85 08              STA  CHAR     ; Zeichen merken
880:   C068 BD 00 02 REMLOOP   LDA  BUFFER,X
890:   C06B F0 E0              BEQ  TAKCHAR   ; Zeilenende, Zeichen so übernehmen
900:   C06D C5 08              CMP  CHAR     ; nächstes '"' oder REM oder DATA
910:   C06F F0 DC              BEQ  TAKCHAR   ; ja
920:   C071 C8        GETCHAR  INY
930:   C072 99 FB 01           STA  BUFFER-5,Y  ; Zeichen übernehmen
940:   C075 E8                 INX
950:   C076 D0 F0              BNE  REMLOOP
960:                  ;
970:   C078 A6 7A     NEXTCMD  LDX  TXTPTR
980:   C07A E6 0B              INC  COUNT    ; Zähler auf nächstes Befehlswort
990:   C07C C8        WEITER   INY
1000:  C07D B9 9D A0           LDA  TABLE-1,Y  ; nächster Buchstabe
```

```
PROFI-ASS 64 V2.0    SEITE 4

1010:  C080 10 FA            BPL  WEITER  ; Wort noch nicht zuende ?
1020:  C082 B9 9E A0         LDA  TABLE,Y
1030:  C085 D0 B5            BNE  TESTNEXT  ; auf nächstes Befehlswort testen
1040:  C087 F0 0F            BEQ  NEWTOK  ; neue Tabelle benutzen
1050:                ;
1060:  C089 BD 00 02 NOTFOUND LDA  BUFFER,X
1070:  C08C 10 BD            BPL  TAKCHAR1  ; Zeichen so übernehmen
1080:                ;
1090:  C08E 99 FD 01 ENDE     STA  BUFFER-3,Y  ; Linkbyte Null für Direktmodus
1100:                ;
1110:  C091 C6 7B            DEC  TXTPTR+1
1120:  C093 A9 FF            LDA  #$FF     ; TXTPTR auf $01FF, BUFFER-1
1130:  C095 85 7A            STA  TXTPTR
1140:  C097 60               RTS
1150:                ;
1160:                ; Verarbeitung der neuen Befehle
1170:  C098 A0 00    NEWTOK   LDY  #0      ; Zeiger auf Beginn der neuen Tabelle
1180:  C09A B9 C3 C0         LDA  NEWTAB,Y  ; erstes Zeichen aus Tabelle holen
1190:  C090 D0 02            BNE  NEWTEST
1200:                ;
1210:  C09F C8       NEWCMP   INY
1210:  C0A0 E8               INX
1220:  C0A1 BD 00 02 NEWTEST  LDA  BUFFER,X  ; Vergleichsroutine für neue
1230:  C0A4 38               SEC             ; Befehlstabelle
1240:  C0A5 F9 C3 C0         SBC  NEWTAB,Y
1250:  C0A8 F0 F5            BEQ  NEWCMP
1260:  C0AA C9 80            CMP  #$80
1270:  C0AC D0 04            BNE  NEXTNEW   ; nächsten neuen Befehl testen
1280:  C0AE 05 0B            ORA  COUNT    ; gefunden
1290:  C0B0 D0 99            BNE  TAKCHAR1  ; unbedingter Sprung
1300:                ;
1310:  C0B2 A6 7A    NEXTNEW  LDX  TXTPTR
```

```
PROFI-ASS 64 V2.0    SEITE 5

1320:  C0B4 E6 0B              INC  COUNT     ; Tokennummer erhöhen
1330:  C0B6 C8         WEITER1 INY
1340:  C0B7 B9 C2 C0           LDA  NEWTAB-1,Y; Zeiger auf nächstes Befehlswort
1350:  C0BA 10 FA              BPL  WEITER1
1360:  C0BC B9 C3 C0           LDA  NEWTAB,Y
1370:  C0BF D0 E0              BNE  NEWTEST   ; Eingabezeile damit vergleichen
1380:  C0C1 F0 C6              BEQ  NOTFOUND  ; Ende der neuen Tabelle
1390:                  ;
1400:  C0C3 52 45 50 NEWTAB    .ASC "REPEAT" ; Tabelle der neuen Befehlsworte
1410:  C0C9 55 4E 54           .ASC "UNTIL"
1420:  C0CE 42 45 46           .ASC "BEFEHL"
1430:  C0D4 00                 .BYT 0        ; Ende der Tabelle
]C000-C0D5
NO ERRORS
```

Mit dieser Routine können wir nun eigene Befehlsworte in Tokens umwandeln. Beim Anlegen der neuen Tabelle mit den Befehlsworten müssen Sie darauf achten, daß das letzte Zeichen jedes Befehls mit gesetztem Bit eingegeben wird. Sie erreichen dies dadurch, indem Sie den letzten Buchstaben geshiftet eingeben. In unserem Assemblerlisting wird dies durch ein kursives Zeichen angedeutet. Durch die Umwandlung in Tokens können Sie die neuen Befehle ebenfalls abgekürzt eingeben, z.B. reP für repeat oder uN anstelle von until.

Mit unserem Verfahren können Sie den neuen Befehlen die Tokens von $CC bis $FE zuweisen. Das sind maximal 51 neuen Befehlsworte. Da diese Tabelle mit einem 8-Bit-Register indiziert wird, darf die Länge der Befehle zusammen nicht länger als 255 Zeichen sein. Das Ende der Tabelle muß durch ein Nullbyte gekennzeichnet werden.

Um unsere neue Routine zu aktivieren, müssen Sie den Vektor $304/$305 auf die Routine richten. Bevor wir dies jedoch machen, wollen wir erst die gegensätzliche Routine schreiben, damit wir Zeilen mit den neuen Befehlen auch listen können. Dazu dient der BASIC-Vektor $306/$307. Über diesen Vektor läuft lediglich das Umsetzen eines Tokens in den Klartext; die organisatorische Arbeit, wie z.B. Zeilenende und neue Zeilennummer ausgeben, nimmt uns die Listroutine schon ab. Sehen wir uns wieder die Interpreterroutine an, die dadurch sehr kurz ausfällt.

```
PROFI-ASS 64 V2.0     SEITE 1

100:   A71A                    .OPT P
110:                 ;
120:                 ; LIST-ROUTINE DES INTERPRETERS
130:                 ;
140:   000F          QUOTFLG   =    15       ; Flag für Hochkommamodus
150:   0049          PNT       =    $49
160:   A09E          TABLE     =    $A09E    ; Befehlstabelle des Interpreters
170:   AB47          CHAROUT   =    $AB47    ; ein Zeichen ausgeben
180:                 ;
190:   A71A                    *=   $A71A
200:   A71A 10 07              BPL  $A6F3    ; kein Interpreterkode, so ausgeben
210:   A71C C9 FF              CMP  #$FF
220:   A71E F0 D3              BEQ  $A6F3    ; Kode für PI, so ausgeben
230:   A720 24 0F              BIT  QUOTFLG   ; Hochkommamodus ?
240:   A722 30 CF              BMI  $A6F3    ; ja, unverändert ausgeben
250:   A724 38                 SEC
260:   A725 E9 7F              SBC  #$7F     ; Offset abziehen
270:   A727 AA                 TAX           ; Kode als Zähler merken
280:   A728 84 49              STY  PNT      ; Zeiger merken
290:   A72A A0 FF              LDY  #-1
300:   A72C CA       NEXT      DEX
```

```
PROFI-ASS 64 V2.0    SEITE 2

A72D F0 08                 BEQ  FOUND    ; X. Befehlswort gefunden ?
A72F C8         LOOP       INY
A730 B9 9E A0              LDA  TABLE,Y
A733 10 FA                 BPL  LOOP     ; Wort noch nicht zuende ?
A735 30 F5                 BMI  NEXT     ; Nächstes Wort
                ;
A737 C8         FOUND      INY
A738 B9 9E A0              LDA  TABLE,Y  ; Buchstaben holen
A73B 30 B2                 BMI  $A6EF    ; letztes Zeichen ?
A73D 20 47 AB              JSR  CHAROUT  ; Zeichen ausgeben
A740 D0 F5                 BNE  FOUND    ; unbedingter Sprung
]A71A-A742
NO ERRORS
```

Die Routine prüft also nach, ob es sich überhaupt um einen Interpreterkode handelt (ist Bit 7 gesetzt?). Der spezielle Kode für Pi wird ebenfalls unverändert ausgegeben. Auch im Hochkommamodus wird zurückverzweigt. Jetzt kommt erst die eigentliche Suche nach dem Befehlswort. Durch Abzug von $7F werden die Interpreterkodes in den Bereich 1 - 76 gebracht. Nun wird die Befehlstabelle durchsucht und am Ende jedes Befehlswortes, das durch das gesetzte Bit sieben erkannt wird, wird die Kodenummer um eins erniedrigt. Ist die Nummer bis auf Null herabgezählt, haben wir das zugehörige Wort in der Tabelle gefunden. Jetzt geben wir alle Zeichen aus, bis wir auf ein Zeichen mit gesetzem Bit 7 stoßen. In diesem Falle verzweigen wir in die LIST-Routine zurück. Dort wird Bit 7 gelöscht und das letzte Zeichen ausgegeben.

Haben wir nun neue Tokens zu listen, so brauchen wir lediglich zu prüfen, ob das Token größer als $CB ist. Ist dies der Fall, so können wir nach dem gleichen Schema das

Befehlswort aus unserer neuen Tabelle suchen, ansonsten lassen wir den Interpreter die Arbeit für die alten Befehle machen.

```
PROFI-ASS 64 V2.0    SEITE 1

100:   C000                  .OPT P,00
110:               ;
120:               ; LIST-ROUTINE FÜR NEUE BEFEHLE
130:               ;
140:   000F        QUOTFLG  =    15
150:   0049        PNT      =    $49
160:   A09E        TABLE    =    $A09E    ; Befehlstabelle
170:   AB47        CHAROUT  =    $AB47    ; Zeichen ausgeben
180:               ;
200:   C000 10 0F           BPL  OUT      ; kein Token, so ausgeben
210:   C002 24 0F           BIT  QUOTFLG  ; Hochkommamodus ?
220:   C004 30 0B           BMI  OUT      ; so ausgeben
230:   C006 C9 FF           CMP  #$FF     ; Pi ?
240:   C008 F0 07           BEQ  OUT      ; so ausgeben
250:   C00A C9 CC           CMP  #$CC     ; neues Token ?
260:   C00C B0 06           BCS  NEWLIST  ; ja
270:               ;
280:   C00E 4C 24 A7        JMP  $A724    ; alte Tokens listen
290:   C011 4C F3 A6 OUT    JMP  $A6F3    ; Byte so ausgeben
300:               ;
310:   C014 38     NEWLIST  SEC
320:   C015 E9 CB           SBC  #$CB     ; Offset abziehen
330:   C017 AA              TAX           ; Kode als Zähler
340:   C018 84 49           STY  PNT
350:   C01A A0 FF           LDY  #-1
360:   C01C CA     NEXT     DEX           ; Wort gefunden ?
370:   C01D F0 08           BEQ  FOUND    ; ja
380:   C01F C8     LOOP     INY
```

```
PROFI-ASS 64 V2.0    SEITE 2

390:   C020 B9 35 C0          LDA  NEWTAB,Y
400:   C023 10 FA             BPL  LOOP     ; Ende des Worts abwarten
410:   C025 30 F5             BMI  NEXT     ; nächstes Wort
420:                    ;
430:   C027 C8          FOUND INY
440:   C028 B9 35 C0          LDA  NEWTAB,Y ; Beehlswort
450:   C02B 30 05             BMI  OLDEND   ; zu Ende ?
460:   C02D 20 47 AB          JSR  CHAROUT  ; Zeichen ausgeben
470:   C030 D0 F5             BNE  FOUND    ; und weiter
480:                    ;
490:   C032 4C EF A6 OLDEND   JMP  $A6EF    ; zur alten Routine
500:                    ;
510:   C035 52 45 50 NEWTAB   .ASC "REPEAT" ; Befehlstabelle
520:   C03B 55 4E 54          .ASC "UNTIL"
530:   C040 42 45 46          .ASC "BEFEHL"
540:   C046 00                .BYT 0
]C000-C047
NO ERRORS
```

Wenn wir den LIST-Vektor $306/$307 auf diese Routine setzen, können wir unsere neuen Befehle auch korrekt listen. Die Befehlstabelle NEWTAB ist natürlich identisch mit der Tabelle in der Routine zur Erzeugung von Tokens und braucht selbstverständlich nur einmal vorhanden zu sein. Bei der praktischen Anwendung sollten Sie die beiden Routinen zusammen assemblieren und gleich ein Initialisierungsprogramm vorsehen, das die beiden Vektoren entsprechend ändert.

Damit die neuen Befehle nun auch vom BASIC-Interpreter verarbeitet werden können, brauchen wir noch Routinen, die die neuen Befehle und Funktionen aufrufen können. Dies geschieht, wie wir bereits wissen, über die Vektoren $308/309

für Befehle und $30A/$30B für Funktionen. Um die Verarbeitung zu vereinfachen, sollten die neuen Befehle so angeordnet werden, daß Befehle und Funktionen jeweils einen Block bilden. Die Routinen zur Einbindung sehen dann so aus, daß zu Beginn geprüft wird, ob das Token im Bereich der neuen Befehle bzw. Funktionen liegt. Man kann dann die Nummer des Tokens als Zeiger in eine Tabelle benutzen, die die Startadressen der zugehörigen Befehle enthält. Das ist genau das Verfahren, das auch der Interpreter benutzt. Wir geben Ihnen nun eine universelle Routine an, die die Verarbeitung neuer Token übernimmt. Sie brauchen vor dem Assemblieren nur den Bereich der neuen Befehle und Funktionen festzulegen und die Startadressen der zugehörigen Routinen in eine Tabelle eintragen.

```
PROFI-ASS 64 V2.0     SEITE 1

100:  C000                      .OPT P1
110:                  ;
120:                  ; EINBINDUNG NEUER TOKENS
130:                  ;
150:  0308            CMDVEK   =  $308    ; Befehlsvektor
160:  030A            FUNVEK   =  $30A    ; Funktionsvektor
170:                  ;
170:  000D            TYPFLAG  =  13      ; Flag numerisch/String
180:  0073            CHRGET   =  $73
190:  0079            CHRGOT   =  CHRGET+6
200:  007A            TXTPTR   =  CHRGOT+1
210:  A7ED            EXECOLD  =  $A7ED   ; alte Befehlsausführung
215:  A7AE            INTER    =  $A7AE   ; Interpreterschleife
217:  AE80            FUNKTOLD =  $AE80   ; alte Funktionsberechnung
218:  AEF1            GETTERM  =  $AEF1   ; Ausdruck in Klammern holen
219:  AD8D            CHECKNUM =  $AD8D   ; Test auf numerisches Ergebnis
220:  0054            JUMP     =  $54     ; Sprungbefehl für Funktionen
```

```
PROFI-ASS 64 V2.0    SEITE 2

300:   00CC           CMDSTART =    $CC      ; erstes Befehlstoken
310:   00E0           CMDEND   =    $E0      ; letztes Befehlstoken
320:                  ;
330:   00E1           FUNSTART =    $E1      ; erstes Funktionstoken
340:   00FE           FUNEND   =    $FE      ; letztes Funktionstoken
350:                  ;
400:   C000 A9 15     INIT     LDA  #<NEWCMD
410:   C002 A0 C0              LDY  #>NEWCMD
420:   C004 8D 08 03           STA  CMDVEK   ; Befehlsvektor
430:   C007 8C 09 03           STY  CMDVEK+1
440:                  ;
450:   C00A A9 3C              LDA  #<NEWFUN
460:   C00C A0 C0              LDY  #>NEWFUN
470:   C00E 8D 0A 03           STA  FUNVEK   ; Funktionsvektor
480:   C011 8C 0B 03           STY  FUNVEK+1
490:   C014 60                 RTS
500:                  ;
510:   C015 20 73 00 NEWCMD    JSR  CHRGET   ; Token holen
510:   C018 20 1E C0           JSR  TESTCMD  ; Befehl ausführen
510:   C01B 4C AE A7           JMP  INTER    ; zurück zur Interpreterschleife
510:                  ;
520:   C01E C9 CC     TESTCMD  CMP  #CMDSTART
530:   C020 90 04              BCC  OLDCMD   ; alter Befehl ?
540:   C022 C9 E1              CMP  #CMDEND+1
550:   C024 90 06              BCC  OKNEW    ; neuen Befehl verarbeiten
560:   C026 20 79 00 OLDCMD    JSR  CHRGOT   ; Flags wieder herstellen
570:   C029 4C ED A7           JMP  EXECOLD  ; und alten Befehl ausführen
580:                  ;
590:   C02C 38        OKNEW    SEC           ; neue Befehle
600:   C02D E9 CC              SBC  #CMDSTART ; Offset abziehen
610:   C02F 0A                 ASL           ; mal 2
620:   C030 AA                 TAX
```

```
630:  C031 BD 6F C0          LDA  CMDTAB+1,X  ; Hi-Byte
640:  C034 48                PHA              ; Rücksprungadresse auf Stack
650:  C035 BD 6E C0          LDA  CMDTAB,X
660:  C038 48                PHA              ; Lo-Byte
670:  C039 4C 73 00          JMP  CHRGET   ; nächstes Zeichen holen
680:                ;
700:  C03C A9 00    NEWFUN   LDA  #0
710:  C03E 85 0D             STA  TYPFLAG ; Typ auf numerisch
720:  C040 20 73 00          JSR  CHRGET   ; Token holen
730:  C043 C9 E1             CMP  #FUNSTART
740:  C045 90 04             BCC  OLDFUN   ; alte Funktion ?
750:  C047 C9 FF             CMP  #FUNEND+1
760:  C049 90 06             BCC  OK1NEW
770:  C04B 20 79 00 OLDFUN   JSR  CHRGOT   ; Flags herstellen
780:  C04E 4C 8D AE          JMP  FUNKTOLD; alte Funktion berechnen
790:                ;
800:  C051 38       OK1NEW   SEC              ; neue Funktion
810:  C052 E9 E1             SBC  #FUNSTART ; Offset abziehen
820:  C054 0A                ASL
830:  C055 48                PHA              ; Zeiger auf Tabelle merken
840:  C056 20 73 00          JSR  CHRGET   ; nächstes Zeichen holen
850:  C059 20 F1 AE          JSR  GETTERM ; Funktionsargument holen
860:  C05C 68                PLA
870:  C05D A8                TAY              ; Zeiger als Index
880:  C05E B9 72 C0          LDA  FUNTAB,Y; Lo-Adresse
890:  C061 85 55             STA  JUMP+1
900:  C063 B9 73 C0          LDA  FUNTAB+1,Y ; Hi-Adresse
910:  C066 85 56             STA  JUMP+2
920:  C068 20 54 00          JSR  JUMP     ; Funktion ausführen
930:  C06B 4C 8D AD          JMP  CHECKNUM  ; Ergebnis auf numerisch testen
940:                ;
950:                ;
```

```
950:    C06E XX XX    CMDTAB   .WOR CMD1-1   ; Tabelle der Befehlsadressen -1
960:    C070 XX XX             .WOR CMD2-1
970:                  ;....
980:    C072 XX XX    FUNTAB   .WOR FUN1     ; Tabelle der Funktionsadressen
990:    C074 XX XX             .WOR FUN2
]C000-C076
NO ERRORS
```

Wenn Sie diese universelle Routine benutzen wollen, brauchen Sie lediglich in Zeile 300 und 310 die Nummer des ersten und letzten neuen Befehlstoken sowie in Zeile 330 und 340 die entsprechenden Nummern für Ihre numerischen Funktionen angeben. Damit die Routine weiß, an welcher Stelle Ihre neuen Befehle stehen, steht von Zeile 950 bis 960 eine Tabelle, die die Adressen der Routinen enthält. Da die Routinen über RTS angesprungen wird, in dem auf den Stack eine Rücksprungadresse gelegt wird, muß bei den Adressen eins abgezogen werden, da beim RTS-Befehl die Rücksprungadresse automatisch um eins erhöht wird.

Bei den Funktionen ist dies nicht erforderlich, da diese über normale JSR-Aufrufe angesprungen werden.

3.5 Die Vektoren des Betriebssystems

Ähnlich wie der BASIC-Interpreter so gehen auch sämtliche wichtige Funktion des Betriebssystem über Sprungvektoren, die wir für unsere eigenen Zwecke abwandeln können. Neben den Hardware-Vektoren IRQ, BRK und NMI, die wir bereits kennengelenrt haben, gehen alle elementaren Ein/Ausgabe-Funktionen über solche Vektoren. Es handelt sich dabei um die Funktionen, die über die Kernalroutinen $FFXX angesprochen werden. Die folgende Tabelle enthält alle diese Vektoren und die Adressen, auf die diese Vektoren nach dem Einschalten zeigen.

Vektor	Adresse	Bedeutung
$0314/$0315	$EA31	IRQ-Vektor
$0316/$0317	$FE66	BRK-Vektor
$0318/$0319	$FE47	NMI-Vektor
$031A/$031B	$F34A	OPEN-Vektor
$031C/$031D	$F291	CLOSE-Vektor
$031E/$031F	$F20E	CHKIN-Vektor
$0320/$0321	$F250	CKOUT-Vektor
$0322/$0323	$F333	CLRCH-Vektor
$0324/$0325	$F157	BASIN-Vektor
$0326/$0327	$F1CA	BSOUT-Vektor
$0328/$0329	$F6ED	STOP-Vektor
$032A/$032B	$F13E	GET-Vektor
$032C/$032D	$FE66	Warmstart-Vektor (unbenutzt)
$032E/$032F	$F4A5	LOAD-Vektor
$0330/$0331	$F5ED	SAVE-Vektor

Im folgenden werden wir die Bedeutung der Vektoren und die Funktion der zugehörigen Routinen kennenlernen. Auf dieser Basis können wir dann eigene Ein-Ausgabe-Routinen schreiben.

OPEN - JSR $FFC0

Diese Routine vollzieht die gleichen Aufgaben, die wir von dem gleichnamigen BASIC-Befehl her schon kennen. Vor dem Aufruf müssen jedoch die Parameter schon versorgt sein. Dazu gibt es zwei Routinen, die diese Aufgaben erledigen.

SETFLS - JSR $FFBA

Diese Routine setzt die Parameter für logische Filenummer, Geräteadresse und Sekundäradresse. Die Parameter werden einfach in den Prozessorregistern übergeben:

```
    LDA LF ; logische Filenummer
    LDX FA ; Geräteadresse
    LDY SA ; Sekundäradresse
    JSR SETFLS ; Parameter setzen
```

Für die Übergabe des Filenamens existiert die Routine SETNAM - JSR $FFBD. Ihr müssen die Länge sowie die Adresse des Filenamens übergeben werden. Wird kein Filename benutzt, so wird als Länge Null benutzt.

```
    LDA #NAME1-NAME ; Länge des Namens
    LDX #< NAME ; Lo-Byte der Adresse
    LDY #> NAME ; Hi-Byte der Adresse
    JSR SETNAM ; Parameter übergeben
    ...
NAME .ASC "FILENAME"
NAME1 = * ; Ende des Namens
```

Wenn diese beiden Routinen ihre Arbeit erledigt haben, kann man die OPEN-Routine aufrufen.

JSR OPEN

Damit wird die logische Datei geöffnet. Um dabei eventuell auftretende Fehler zu erkennen, hat man sich folgendes Verfahren ausgedacht. Das Carryflag wird dabei als Fehlerflag benutzt. Ist es nach dem Aufruf der Routine gelöscht, so wurde die Routine fehlerfrei ausgeführt. Trat jedoch ein Fehler auf, so wird das Carryflag gesetzt und der Akku enthält die Fehlernummer. Diese Fehlernummern haben die folgende Bedeutung:

Nr.	Bedeutung
0	Abbruch durch STOP-Taste
1	too many files
2	file open
3	file not open
4	file not found
5	device not present
6	not input file
7	not output file
8	missing filename
9	illegal device number
240	RS 232 Open/Close

Nach dem Aufruf einer Kernalroutine sollte daher das Carryflag getestet werden, um den Fehlerstatus zu testen.

```
    JSR OPEN ; File öffnen
    BCC OK   ; alles OK ?
    JMP ERROR
OK ...
```

Die Fehlernummern entsprechen den gleichnamigen Fehlermeldungen, die wir schon von BASIC her kennen. Eine neue

Fehlernummer tritt beim OPEN oder CLOSE mit der Gerätenummer 2 auf, der RS-232-Schnittstelle. Wie Sie vielleicht wissen, werden beim Öffnen eines RS-232-Kanals zwei Puffer zu je 256 Bytes für Ein- und Ausgabe angelegt. Diese Puffer werden ans obere Ende des BASIC-Bereichs gelegt. Normalerweise wird also das BASIC-Ende von $A000 auf $9E00 zurückgelegt. Da in diesem Bereich aber normalerweise Strings abgelegt werden, stehen diese nach einem RS-232 Open nicht mehr zur Verfügung. Um dem BASIC-Interpreter diese Situation mitzuteilen, wird das Fehlerflag gesetzt und die Fehlernummer 240 übergeben. Der BASIC-Interpreter führt daraufhin einen CLR-Befehl aus; er löscht also sämtliche Variablen. Bei einem CLOSE-Befehl werden diese Puffer wieder freigegeben und die Variablen werden ebenfalls gelöscht. Wenn Sie also die RS-232 Schnittstelle in Ihren BASIC-Programmen benutzen, sollten Sie den OPEN-Befehl als ersten in Ihrem Programm benutzen und den CLOSE-Befehl als letzten. Dadurch ist sichergestellt, daß während des Programmablaufs keine Variablen gelöscht werden. Hier wäre auch schon ein Ansatzpunkt, unsere OPEN-Routine zu ändern. Wir könnten beim Öffnen der RS 232 Schnittstelle die Puffer einfach in den Bereich ab $C000 legen. Dadurch wird der BASIC-Bereich nicht beeinträchtigt und der CLR-Befehl kann entfallen.

Auch bei den noch zu besprechenden Ein/Ausgabe-Routinen wird das Carryflag als Fehlerflag benutzt und der Akku enthält die Fehlernummer.

Das Betriebssystem besitzt sogar eine eigene Routine zur Ausgabe von Fehlermeldungen. Die Ausgabe geschieht in der Form, daß eine Meldung

I/O ERROR #X

generiert wird, wobei X die Fehlernummer ist (1 bis 9). Das Programm wird dabei jedoch nicht abgebrochen. Die Fehlerausgabe können wir dadurch aktivieren, daß wir die Routine SETMSG - JSR $FF90 mit einem Wert von $40 im Akku (Bit 6 gesetzt) aufrufen. Abschalten können wir die Fehlermeldungen, indem wir der Routine SETMSG den Wert Null übergeben.

Eine weitere Funktion der Routine SETMSG ist die Unterscheidung zwischen Programm und Direktmodus. Dafür ist Bit 7 zuständig. Ist Bit 7 gelöscht, so wird dadurch der Programmmodus gekennzeichnet und Statusmeldungen des Betriebssystems wie 'SEARCHING FOR', 'LOADING' und 'SAVING' werden unterdrückt.

CLOSE - JSR $FFC3

Die CLOSE-Routine benötigt nur einen Parameter, die logische Filenummer. Sie wird im Akku übergeben.

```
LDA LF
JSR CLOSE
```

Bei der CLOSE-Routine können keine Fehlermeldungen auftreten. Eine Ausnahme bildet wieder das Schließen eines RS 232-Kanals. Hierbei wird der Puffer wieder freigegeben und BASIC führt anschließend einen CLR-Befehl durch. Der Versuch, eine nicht geöffnete Datei zu schließen, erzeugt keine Fehlermeldung.

CHKIN - JSR $FFC6

Dieser Befehl dient dazu, die Eingabe von der Tastatur auf eine geöffnete Datei umzulenken. Wenn Sie z.B. Daten von der Diskette lesen wollen, so müssen Sie die Datei erst öffnen und können dann mit CHKIN diese Datei als Eingabe benutzen. Beim Aufruf muß die logische Datei im X-Register übergeben werden.

```
LDX LF
JSR CHKIN
```

Auch hierbei werden wieder Fehler durch das gesetzte Carryflag erkannt. War die Datei vorher nicht geöffnet, erhalten wir 'file not open'; versuchen Sie von einer Banddatei zu lesen, die zum Schreiben geöffnet war, wird ein 'not input file' error angezeigt. Die eigentliche Eingabe geschieht dann über die Routine BASIn, doch dazu später.

CKOUT - JSR $FFC9

Was die Routine CHKIN zur Vorbereitung der Eingabe ist, das ist CKOUT für die Ausgabe. Damit läßt sich die Ausgabe auf eine vorher geöffnete Datei lenken. Die CKOUT-Routine entspricht dem BASIC-Befehl CMD. Die logische Filenummer wird wieder im X-Register übergeben.

```
LDX LF
JSR CKOUT
```

Die möglicherweise auftretenden Fehler entsprechen denen von CHKIN. Beim Versuch, in eine zum Lesen geöffnete Banddatei zu schreiben, erhalten wir 'not output file'. Die Ausgabe geschieht dann später mit BSOUT.

BASIN - JSR $FFCF

Die Routine läßt sich mit dem INPUT-Befehl in BASIC vergleichen. Wenn Sie vorher nicht mit CHKIN die Eingabe auf eine Datei umgeleitet haben, so können Sie mit BASIN Zeichen von der Tastatur bzw. dem Bildschirm holen. Wenn Sie innerhalb eines Maschinenprogramms BASIN aufrufen, so erscheint der Cursor auf dem Bildschirm und Sie können solange Zeichen eingeben, bis Sie Return drücken. Die Routine BASIN gibt im Akku das erste eingegebene Zeichen zurück. Jeder weitere Aufruf von BASIN holt ein weiteres Zeichen, bis zum Return (CHR$(13)). Dabei können Sie den Bildschirmeditor voll benutzen. Wollen Sie jedoch Zeichen aus einer geöffneten Datei entsprechend dem INPUT#-Befehl, so müssen Sie vorher die bereits besprochene Routine CHKIN aufrufen, die die Eingabe auf diese Datei umleitet. Die BASIN-Routine holt dann bei jedem Aufruf ein Zeichen aus der geöffneten Datei und stellt es im Akku zur Verfügung.

BSOUT - JSR $FFD2

Mit der BSOUT-Routine können wir Zeichen ausgeben. Dabei wird das Zeichen im Akku auf dem Bildschirm ausgegeben. Der Akku muß den ASC-Wert des auszugebenden Zeichens enthalten, z.B.

```
LDA #$41
JSR BSOUT
```

Damit wird das Zeichen mit dem ASC-Wert $41 gleich 65 ausgegeben, das ist das "A". Sie können damit auch Steuerzeichen oder Farbkodes ausgeben, ganz analog wie dies in BASIC mit PRINT CHR$(X); geschieht. Eine evtl. neue Zeile, wie dies in BASIC durch den PRINT-Befehl ohne nachfolgendes

Semikolon möglich ist, muß in Maschinensprache explizit programmiert werden.

```
LDA #13 ; Carriage Return
JSR BSOUT ; ausgeben
```

Wollen Sie dagegen die Zeichen nicht auf den Bildschirm, sondern auf den Drucker oder auf Diskette ausgeben, so müssen Sie vorher eine entsprechende Datei öffnen und die Routine CKOUT benutzen. Damit wird die Ausgabe auf die Datei gelegt und sämtliche Aufrufe von BSOUT geben das Zeichen nun nicht mehr auf den Bildschirm, sondern in die Datei. Als Fehlermeldung kann z.B. 'device not present' kommen, wenn das Gerät auf dem seriellen Bus nicht antwortet.

CLRCH - JSR $FFCC

Die Routine CLRCH hat die gegensätzliche Funktion von CHKIN und CKOUT. Während diese Routine die Ein- bzw. Ausgabe auf eine logische Datei umleiten, werden mit CLRCH wieder die Standard-Ein/Ausgabe-Geräte gesetzt, also wieder Tastatur bzw. Bildschirm. Wenn Sie also 10 Zeichen aus der logischen Datei 2 von der Floppy holen wollen, sieht der entsprechende Programmabschnitt so aus:

```
     LDX #2 ; logische Filenummer
     JSR CHKIN ; Eingabe von Datei #2
     LDY #0
LOOP JSR BASIN ; Zeichen von Floppy holen
     STA STORE,Y ; und abspeichern
     INY
     CPY #10 ; schon 10 Zeichen ?
     BNE LOOP ; nein
     JSR CLRCH ; wieder auf Standardeingabe
```

Vor der Anwendung dieses Programmabschnitts muß die logische Datei 2 geöffnet sein. Dann wird mit CHKIN auf Eingabe von Datei umgeschaltet, zehn Zeichen mit BASIN geholt und abgespeichert, bevor mit CLRCH wieder auf Standardeingabe von Tastatur umgeschaltet wird. Die Datei bleibt dabei weiterhin offen; das Schließen muß explizit mit CLOSE geschehen.

GET - JSR $FFE4

Diese Routine entspricht der GET-Routine von BASIC. Sie können damit ein Zeichen von der Tastatur holen. Ist zum Zeitpunkt des Aufrufs keine Taste gedrückt, so bekommen Sie den Kode Null im Akku zurück analog zu BASIC, wo Sie einen Leerstring erhalten, wenn keine Taste gedrückt ist. Das Warten auf einen Tastendruck könnte man also so formulieren:

```
LOOP  JSR GET
      BEQ LOOP
```

Die Schleife wartet also solange, bis eine Taste gedrückt wird. Den GET-Befehl können Sie auch auf eine logische Datei beziehen. Dazu muß wie bei BASIN vorher mit CHKIN die logische Datei bestimmt werden. Der GET-Befehl auf eine Datei arbeitet analog zur BASIN-Routine, so daß wir auch diese dazu benutzen können. Nach dem GET auf eine logische Datei ist der Aufruf von JSR CLRCH erforderlich, um die Standardeingabe wieder zu aktivieren.

CLALL - JSR $FFE7

Diese Routine hat die gleiche Aufgabe wie CLRCH. Zusätzlich wird jedoch die Zahl der offenen Dateien auf Null gesetzt. Dies kommt für den Rechner einem Schließen aller Dateien gleich. Dabei wird jedoch nicht die zugehörige CLOSE-Routine

aufgerufen. Eine zum Schreiben geöffnete Datei auf der Floppy wird dadurch nicht ordnungsgemäß geschlossen. Diese Routine wird vom BASIC-Interpreter beim RUN-Befehl aufgerufen.

LOAD - JSR $FFD5

Das ist die LOAD-Routine des Betriebssystems. Vor dem Aufruf dieser Routine müssen Geräteadresse, Sekundäradresse und Filename gesetzt werden. Dies kann mit den Routinen SETFLS und SETNAM geschehen, die beim OPEN-Befehl besprochen wurden. Abhängig von der Sekundäradresse kann ein Programm an die Adresse geladen werden, die auf Diskette oder Datasette vermerkt ist oder an eine Adresse, die der LOAD-Routine zu übergeben ist. Bei einer Sekundäradresse von Null wird an die Adresse geladen, die im X-(lo) und Y-Register (hi) übergeben wird. Der Akkuinhalt entscheidet, ob ein Laden oder lediglich ein Verify durchgeführt wird.

```
LDA #0 ; Flag für LOAD
LDX #< ADRESSE ; Startadresse
LDY #> ADRESSE
JSR LOAD
STX ENDADR ; Endadresse lo
STY ENDADR+1 ; hi
```

Für den Fall, daß die Sekundäradresse Null ist, wird das Programm an die Adresse geladen, die in ADRESSE angegeben wird. Die Endadresse des geladenen Programms wird von der LOAD-Routine in X und Y zur Verfügung gestellt. Soll das Programm nicht geladen, sondern lediglich mit dem im Speicher stehenden verglichen werden, so muß im Akku eine Eins übergeben werden.

```
    LDA #1 ; Flag für VERIFY
    JSR LOAD
```

Wenn die Sekundäradresse Eins ist, wird an die Adresse geladen, die im Programm selbst abgespeichert ist und wir brauchen keine Startadresse in X und Y angeben. Bei Verify wird eine Nichtübereinstimmung durch einen Statuswert (Adresse $90) von ungleich Null gekennzeichnet. Bit 6 (Wert 64) muß dabei jedoch ausgeklammert werden, da dadurch das Programmende erkannt wird.

```
   LDA STATUS
   AND #%10111111 ; EOF-Bit maskieren
   BEQ OK
   JMP ERROR
OK ...
```

SAVE - JSR $FFD8

Mit der SAVE-Routine ist es möglich, einen beliebigen Speicherbereich auf ein Peripheriegerät abzuspeichern. Mit den Routinen SETFLS und SETNAM müssen vorher wieder Geräteadresse und Filenamen festgelegt werden. Der Routine selbst müssen Start- und Endadresse+1 des abzuspeichernden Bereichs angegeben werden. Die Endadresse plus eins muß dazu in X- und Y-Register stehen. Im Akku muß ein Zeiger auf die Zeropageadresse stehen, an der Lo- und Hi-Byte der Startadresse stehen. Wollen wir z.B. den Bereich von $1234 bis $1FFF abspeichern, kann der Aufruf so aussehen:

```
   LDA #< $1234
   STA START
   LDA #> $1234
   STA START+1
```

```
LDX #< $1FFF+1
LDY #> $1FFF+1
LDA #START
JSR SAVE
```

Zuerst wird also die Startadresse in den Zeropagedaressen START und START+1 abgelegt. Die Endadresse plus eins wird in X (lo) und Y-Register (hi) hinterlegt. Der Akku wird mit der Adresse von START geladen. Beachten Sie dabei die unmittelbare Adressierung, da die Adresse selbst, nicht deren Inhalt, gemeint ist.

Als Fehlermeldungen können 'device not present', 'missing filename' auftreten,wenn auf Diskette abgespeichert werden sollte oder 'illegal device number' beim Versuch auf Tastatur, Bildschirm oder RS 232 abzuspeichern.

Ehe wir versuchen, eigene Ein-Ausgabe-Routinen zu schreiben, gehen wir noch kurz auf die Arbeitsweise der Kernalroutinen im Betriebssystem ein.

OPEN

Beim OPEN-Befehl werden die Parameter für logische Filenummer, Geräteadresse und Sekundäradresse in je eine Tabelle eingetragen. Diese Tabelle umfaßt zehn Positionen. Beim Versuch, mehr als 10 Files zu öffnen, wird die Fehlermeldung 'too many files' generiert. Das weitere Vorgehen ist abhängig von der Geräteadresse. Handelt es sich um Tastatur (=0) oder Bildschirm (=3), so wird ein eventueller Filename nicht beachtet und die Routine beendet. Bei der Datasette (=1) wird abhängig von der Sekundäradresse eine Banddatei zum Lesen (Sek.-Adr =0) oder zum Schreiben

(Sek.-Adr =1) geöffnet. Sekundäradresse 2 führt ebenfalls zum Öffnen einer Schreibdatei und hat nur beim CLOSE-Befehl eine unterschiedliche Handhabung zur Folge. Beim Lesen wird die Banddatei mit dem im OPEN-Befehl angegegebenen Namen gesucht. Wurde kein Name angegeben, so wird die erste Datei geöffnet, die gefunden wird. Beim Schreiben wird eine Datei mit dem geforderten (oder keinem) Namen geöffnet. Handelte es sich bei der Geräteadresse um 2, so wird die RS 232 Übertragung vorbereitet, Wie bereits erwähnt, werden zwei Puffer zu je 256 Bytes für Ein- und Ausgabe vom oberen Ende des BASIC-Speichers abgezweigt. Die Sekundäradresse wird hierbei nicht beachtet. Die ersten beiden Zeichen des 'Filenamens' werden nach $293 und $294 kopiert. Aus diesen Parametern wird die Anzahl der zu übertragenden Bits (5 bis 8) berechnet und nach $298 gespeichert. Aus der Baudrate im ersten Zeichen des Filenamens werden über eine Tabelle die entsprechenden Werte ermittelt, mit denen die Timer in der CIA 2 geladen werden müssen, und nach $295/$296 abgespeichert. War X-Line Handshake angegeben, wird geprüft, ob das Signal DSR (Data Set Ready) vorhanden ist. Beim Fehlen dieses Signals wird das entsprechende Bit im RS 232 Status ($297) gesetzt. Ansonsten wird der Status beim OPEN-Befehl immer gelöscht. Bei Geräteadressen von über 3 ist der serielle Bus angesprochen. Fehlen Sekundäradresse und Filename, z.B. OPEN 1,4 für den Drucker, so geschieht lediglich der Eintrag in die Tabelle des Rechners. Das Fehlen einer Sekundäradresse muß der Routine SETFLS durch einen negativen Wert ($FF) für die Sekundäradresse mitgeteilt werden. Ansonsten wird der OPEN-Befehl über den seriellen Bus geschickt. Dazu wird, nachdem das Gerät mit LISTEN adressiert ist, die Sekundäradresse plus $F0 geschickt. Dies interpretiert das angeschlossene Gerät als OPEN-Befehl. War noch ein Filename angegeben, so wird er anschließend geschickt, ehe die Übertragung mit UNLISTEN beendet wird.

CLOSE

Der CLOSE-Befehl beendet die Übertragungen und löscht die entsprechenden Tabelleneinträge des Rechners. Hierbei wird wieder nach der Gerätenummer unterschieden. Bei Files auf Tastatur und Bildschirm geschieht nichts weiter. Soll eine Banddatei geschlossen werden, ist das Vorgehen abhängig von der Sekundäradresse. War die Datei zum Lesen geöffnet (Sek.-Adr. 0), braucht nichts weiter zu geschehen. Beim Schreiben wird der augenblickliche Inhalt des Kassettenpuffers noch auf Band geschrieben. Bei Sekundäradresse 2 wird zusätzlich noch ein EOT-Block ('End of Tape') geschrieben. Bei einer RS 232 Übertragung werden die Aktivitäten abgebrochen und die beiden Puffer wieder freigegeben. Soll ein File auf dem seriellen Bus geschlossen werden, so sendet der Rechner, falls eine Sekundäradresse angegeben war, diese Sekundäradresse plus $E0, was als CLOSE-Befehl interpretiert wird.

CHKIN

Wenn die Eingabe auf eine Datei umgeleitet werden soll, so ermittelt der Rechner aus der logischen Filenummer die zugehörigen Geräte- und Sekundäradresse und macht sein weiteres Vorgehen davon abhängig. Bei der Datasette wird geprüft, ob es sich um eine Lesedatei (Sek.-Adr. 0) handelt, sonst wird die Fehlermeldung 'not input file' generiert. Bei Geräten auf dem seriellen Bus wird ein TALK-Befehl und anschließend die Sekundäradresse gesandt. Dadurch ist das Gerät bereit, Daten zu senden. Unabhängig vom Gerät wird die Gerätenummer gespeichert, von der nun sämtliche Eingaben erwartet werden, bis mit CLRCH wieder auf die normale Eingabe zurückgeschaltet wird.

CKOUT

Der CKOUT-Befehl funktioniert analog zum CHKIN-Befehl. Bei der Datasette wird auf eine Sekundäradresse größer als Null geprüft (sonst gibt es 'not output file'). Auf dem seriellen Bus wird ein LISTEN-Befehl und anschließend die Sekundäradresse gesandt. Dadurch ist das angeschlossene Gerät bereit, Daten zu empfangen.

BASIN

Hier wird abhängig vom gerade aktiven Gerät, das mit CHKIN ausgewählt wurde, ein Zeichen entweder von Tastatur, der Datasette, der RS 232 Schnittstelle oder dem seriellen Bus geholt und im Akku zur Verfügung gestellt.

BSOUT

Diese Routine sendet das Zeichen im Akku an das vorher mit CKOUT bestimmte Gerät. Als Standardgerät dient hier der Bildschirm.

CLRCH

Der CLRCH-Befehl hebt die mit CHKIN oder CKOUT getroffene Wahl des Ein- oder Ausgabegerätes wieder auf. Dazu werden in den entsprechenden Adressen wieder die Werte 0 für Tastatureingabe bzw. 3 für Bildschirmausgabe eingetragen. Waren vorher Geräte am seriellen Bus aktiv, so wird noch UNTALK-bzw.UNLISTEN-Befehlgesandt,umdenangeschlossenen Geräten das Ende der Übertragung anzuzeigen.

3.6 Druckerspooling

Als Beispiel für die Benutzung der Ein-Ausgabe-Vektoren des Betriebssystems wollen wir eine Druckerschnittstelle realisieren, die den Userport zur Centronicsschnittstelle umfunktioniert und dabei ein Druckerspooling realisiert.

Von Spooling spricht man dann, wenn die Ausgabe der Zeichen an den Drucker im Hintergrund unabhängig vom anderen Programmablauf geschieht. Aus dieser Beschreibung wird Ihnen sicher schon klar, daß es sich dabei um ein Interruptprogramm handeln muß. Damit die normale PRINT-Ausgabe nicht bei jedem Zeichen warten muß, bis der Drucker bereit ist, ein Zeichen anzunehmen, schreiben wir das Zeichen in einen Puffer. Das Interruptprogramm, das wir in den normalen Systeminterrupt einbinden, schaut nun jedesmal nach, ob noch Zeichen im Puffer stehen. Wenn dies der Fall ist und der Drucker ist bereit, ein Zeichen anzunehmen, werden solange Zeichen an den Drucker geschickt bis entweder der Drucker momentan keine Zeichen annehmen kann oder alle Zeichen ausgegeben sind.

```
PROFI-ASS 64 V2.0    SEITE 1

100:  CC00              .OPT P,00
110:              ;
120:              ; DRUCKERSPOOLING
130:              ;
140:              ; I/O-Vektoren
150:  031A        OPEN     =   $31A      ; OPEN-Vektor
160:  031C        CLOSE    =   $31C      ; CLOSE-Vektor
190:  0326        BSOUT    =   $326      ; BSOUT-Vektor
200:              ;
210:  00F7        WPNT     =   $F7       ; Schreibzeiger in Puffer
220:  00F9        RPNT     =   $F9       ; Lesezeiger in Puffer
230:              ;
240:  0098        NRFLS    =   $98       ; Anzahl der offenen Files
250:  00B8        LF       =   $B8       ; logische Filenummer
260:  00BA        FA       =   $BA       ; Geräteadresse
270:  00B9        SA       =   $B9       ; Sekundäradresse
280:  0259        LFTAB    =   $259      ; Tabelle logische Filenummern
290:  0263        FATAB    =   LFTAB+10 ;   "    Geräteadressen
300:  026D        SATAB    =   FATAB+10 ;   "    Sekundäradressen
310:  009E        CHAR     =   $9E       ; auszugebendes Zeichen
320:  0001        KONFIG   =   1         ; Speicheraufteilung
330:  009A        OUTDEV   =   $9A       ; Gerätenummer für Ausgabe
340:  0314        IRQVEK   =   $314      ; IRQ-Vektor
350:  EA31        IRQALT   =   $EA31     ; alte IRQ-Routine
360:              ;
370:  F34A        OPENOLD  =   $F34A
380:  F1CA        BSOUTOLD =   $F1CA
390:  F31F        SETPARA  =   $F31F
400:  F314        SUCHLF   =   $F314
410:  F30F        SUCHLFX  =   $F30F
420:  F2A1        OLDCLOSE =   $F2A1
430:  F2F1        CONTCLS  =   $F2F1
```

```
440:   F6FE            FILEOPEN =   $F6FE
450:   F64B            TOOMANY  =   $F64B
460:   F291            CLOSEOLD =   $F291
470:   DD00            CIA      =   $DD00     ; CIA 2
480:   DD00            PORTA    =   CIA       ; PA2 für Strobe
490:   DD01            PORTB    =   CIA+1     ; Port B für Daten
500:   DD03            RICHTUNG =   CIA+3     ; Datenrichtungsregister
510:   DD0D            ICR      =   CIA+13    ; Interrupt Control Register
520:                   ;
530:   E000            PUFFER   =   $E000     ; Druckerpuffer unter Kernal
540:                   ;
550:   CC00                     *=  $CC00
560:   CC00 A9 0B      INIT     LDA #< OPENNEW
570:   CC02 A0 CC               LDY #> OPENNEW
580:   CC04 80 1A 03            STA OPEN      ; OPEN-Vektor neu setzen
590:   CC07 8C 1B 03            STY OPEN+1
600:   CC0A 60                  RTS
610:                   ;
620:   CC0B A6 B8      OPENNEW  LDX LF        ; Logische Filenummer
630:   CC00 F0 05               BEQ ERROR     ; null nicht erlaubt
640:   CC0F 20 0F F3            JSR SUCHLFX   ; Filedaten suchen
650:   CC12 D0 03               BNE OK2       ; nicht gefunden, ok
660:   CC14 4C FE F6   ERROR    JMP FILEOPEN  ; sonst 'file open' error
670:   CC17 A6 98      OK2      LDX NRFLS     ; Anzahl der offenen Files
680:   CC19 E0 0A               CPX #10
690:   CC1B 90 03               BCC OK        ; weniger als 10, ok
700:   CC1D 4C 4B F6            JMP TOOMANY   ; 'too many files'
710:   CC20 A5 BA      OK       LDA FA        ; Gerätenummer
720:   CC22 C9 04               CMP #4        ; gleich 4 ?
730:   CC24 F0 03               BEQ SPOOL     ; ja, Spooling
740:   CC26 4C 4A F3            JMP OPENOLD
750:   CC29 E6 98      SPOOL    INC NRFLS     ; Anzahl erhöhen
```

```
760:   CC2B 90 63 02        STA  FATAB,X  ; Geräteadresse in Tabelle
770:   CC2E A5 B8           LDA  LF
780:   CC30 90 59 02        STA  LFTAB,X ; Logische Filenummer
790:   CC33 A9 FF           LDA  #-1
800:   CC35 9D 6D 02        STA  SATAB,X ; keine Sekundäradresse
810:   CC38 A9 E0           LDA  #> PUFFER
820:   CC3A 85 F8           STA  WPNT+1   ; Schreib-Zeiger
830:   CC3C 85 FA           STA  RPNT+1   ; und Lese-Zeiger
840:   CC3E A9 00           LDA  #0       ; auf Pufferbeginn
850:   CC40 85 F7           STA  WPNT
860:   CC42 85 F9           STA  RPNT
870:   CC44 A9 FF           LDA  #$FF
870:   CC46 80 03 DD        STA  RICHTUNG  ; User-Port auf Ausgabe
880:   CC49 AD 00 DD        LDA  PORTA
890:   CC4C 09 04           ORA  #%100    ; Strobe hi
900:   CC4E 8D 00 DD        STA  PORTA
910:   CC51 A9 B5           LDA  #< BSOUTNEW
920:   CC53 A0 CC           LDY  #> BSOUTNEW
930:   CC55 8D 26 03        STA  BSOUT    ; BSOUT-Vektor auf neue Routine
940:   CC58 8C 27 03        STY  BSOUT+1
950:   CC5B A9 DD           LDA  #< CLOSENEW
960:   CC5D A0 CC           LDY  #> CLOSENEW
970:   CC5F 8D 1C 03        STA  CLOSE    ; CLOSE-Vektor auf neue Routine
980:   CC62 8C 1D 03        STY  CLOSE+1
990:   CC65 A9 73           LDA  #< SPOOLING
1000:  CC67 A0 CC           LDY  #> SPOOLING
1010:  CC69 78              SEI
1020:  CC6A 8D 14 03        STA  IRQVEK   ; IRQ-Vektor aus Spool-Routine
1030:  CC6D 8C 15 03        STY  IRQVEK+1
1040:  CC70 58              CLI
1040:  CC71 18              CLC           ; Fehlerflag löschen
1050:  CC72 60              RTS
```

```
1060:                  ;
1070:  CC73 A5 01      SPOOLING LDA  KONFIG
1080:  CC75 48                  PHA
1090:  CC76 A9 35               LDA  #$35     ; RAM auswählen
1100:  CC78 85 01               STA  KONFIG
1110:  CC7A A5 F9      TESTNEXT LDA  RPNT     ; Schreib- mit Lesezeiger
1120:  CC7C C5 F7               CMP  WPNT     ; vergleichen
1130:  CC7E D0 06               BNE  SENDCHAR  ; ungleich, dann Zeichen ausgeben
1140:  CC80 A5 FA               LDA  RPNT+1
1150:  CC82 C5 F8               CMP  WPNT+1
1160:  CC84 F0 29               BEQ  EXIT
1170:  CC86 A9 10      SENDCHAR LDA  #%10000 ; Bit-Maske für FLAG-Eingang
1180:  CC88 2C 0D DD            BIT  ICR      ; Drucker bereit ?
1190:  CC8B F0 22               BEQ  EXIT     ; nein
1200:  CC8D A0 00               LDY  #0
1210:  CC8F B1 F9               LDA  (RPNT),Y  ; auszugebendes Zeichen
1220:  CC91 80 01 DD            STA  PORTB    ; auf den Port geben
1230:  CC94 AD 00 DD            LDA  PORTA
1240:  CC97 29 FB               AND  #%11111011  ; Strobe lo
1250:  CC99 8D 00 DD            STA  PORTA
1260:  CC9C 09 04               ORA  #%00000100  ; und wieder hi
1270:  CC9E 8D 00 DD            STA  PORTA
1280:  CCA1 E6 F9               INC  RPNT
1280:  CCA3 D0 D5               BNE  TESTNEXT ; Lesezeiger erhöhen
1290:  CCA5 E6 FA               INC  RPNT+1
1290:  CCA7 D0 D1               BNE  TESTNEXT
1300:  CCA9 A9 E0               LDA  #> PUFFER
1310:  CCAB 85 FA               STA  RPNT+1
1320:  CCAD D0 CB               BNE  TESTNEXT  ; nächstes Zeichen senden
1330:                  ;
1340:  CCAF 68         EXIT     PLA
1350:  CCB0 85 01               STA  KONFIG   ; alte Speicheraufteilung
```

```
PROFI-ASS 64 V2.0    SEITE 5

1360:  CCB2 4C 31 EA          JMP  IRQALT   ; zum alten IRQ
1370:                ;
1380:  CCB5 48       BSOUTNEW PHA           ; Zeichen merken
1390:  CCB6 A5 9A             LDA  OUTDEV   ; Geräteadresse
1400:  CCB8 C9 04             CMP  #4       ; gleich 4 ?
1410:  CCBA F0 04             BEQ  OK1      ; ja
1420:  CCBC 68                PLA
1420:  CCBD 4C CA F1          JMP  BSOUTOLD ; zur alten Ausgabe
1430:  CCC0 68       OK1      PLA           ; Zeichen zurück
1440:  CCC1 85 9E             STA  CHAR     ; und merken
1450:  CCC3 98                TYA
1450:  CCC4 48                PHA           ; Y retten
1460:  CCC5 A5 9E             LDA  CHAR     ; Zeichen
1470:  CCC7 A0 00             LDY  #0
1480:  CCC9 91 F7             STA  (WPNT),Y  ; in Puffer schreiben
1490:  CCCB E6 F7             INC  WPNT
1500:  CCCD D0 08             BNE  NOINC    ; Pufferzeiger erhöhen
1510:  CCCF E6 F8             INC  WPNT+1
1520:  CCD1 D0 04             BNE  NOINC
1530:  CCD3 A9 E0             LDA  #> PUFFER  ; Pufferzeiger auf Anfang
1540:  CCD5 85 F8             STA  WPNT+1
1550:  CCD7 68       NOINC    PLA
1560:  CCD8 A8                TAY           ; Y zurück
1570:  CCD9 A5 9E             LDA  CHAR
1580:  CCDB 18       FERTIG   CLC           ; Fehlerflag löschen
1580:  CCDC 60                RTS
1590:                ;
1600:  CCDD 20 14 F3 CLOSENEW JSR  SUCHLF   ; Filedaten suchen
1610:  CCE0 D0 F9             BNE  FERTIG   ; kein File offen, fertig
1620:  CCE2 20 1F F3          JSR  SETPARA  ; Fileparameter holen
1630:  CCE5 8A                TXA
1630:  CCE6 48                PHA           ; X-Register retten
```

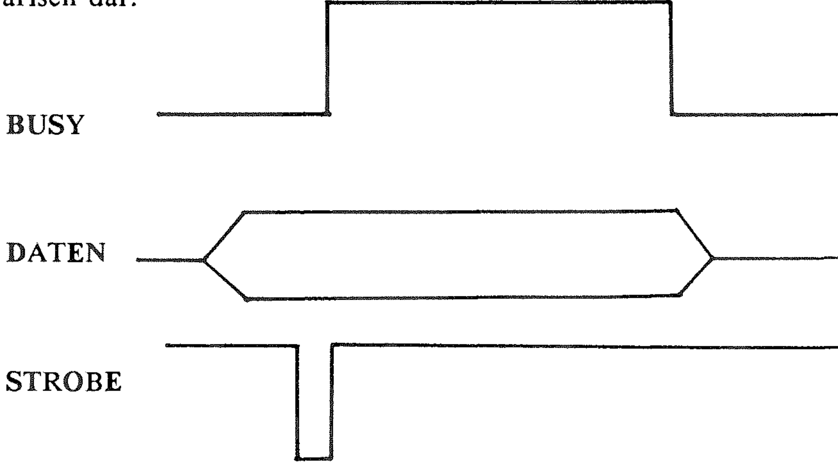
```
PROFI-ASS 64 V2.0    SEITE 6

1640:  CCE7 A5 BA              LDA  FA       ; Geräteadresse
1650:  CCE9 C9 04              CMP  #4       ; 4 ?
1660:  CCEB F0 03              BEQ  CLOSE1
1670:  CCED 4C A1 F2           JMP  OLDCLOSE ; alte CLOSE-Routine
1680:  CCF0 A9 CA     CLOSE1   LDA  #< BSOUTOLD
1690:  CCF2 A2 F1              LDX  #> BSOUTOLD
1700:  CCF4 8D 26 03           STA  BSOUT    ; Vektor auf alte BSOUT-Routine
1710:  CCF7 8E 27 03           STX  BSOUT+1
1720:  CCFA A9 91     CLOSE1   LDA  #< CLOSEOLD
1730:  CCFC A2 F2              LDX  #> CLOSEOLD
1740:  CCFE 80 1A 03           STA  CLOSE    ; Vektor auf alte CLOSE-Routine
1750:  CD01 8E 1B 03           STX  CLOSE+1
1760:  CD04 A9 31              LDA  #< IRQALT
1770:  CD06 A2 EA              LDX  #> IRQALT
1780:  CD08 78                 SEI
1790:  CD09 8D 14 03           STA  IRQVEK   ; alten IRQ wiederherstellen
1800:  CD0C 8E 15 03           STX  IRQVEK+1
1810:  CD0F 58                 CLI
1820:  CD10 4C F1 F2           JMP  CONTCLS  ; CLOSE normal beenden
]CC00-CD13
NO ERRORS
```

Bevor wir zur Besprechung der Routinen kommen, soll zum besseren Verständnis der Druckerausgabe kurz die Funktionsweise der Centronicsschnittstelle beschrieben werden.

Eine Centronicsschnittstelle ist eine Parallelschnittstelle, d.h. es werden immer 8 Bit parallel, also ein komplettes Byte gleichzeitig übertragen. Damit sich Rechner und Drucker über den Zeitpunkt der Übertragung einigen können, werden noch zwei sogenannte Handshakeleitungen benutzt. Die eine Leitung

mit dem Namen STROBE wird vom Rechner bedient. Die Leitung liegt im Ruhezustand auf einem Hi-Pegel. Will der Rechner ein Byte zum Drucker schicken, so legt er die Daten auf die Leitung und zeigt dem Drucker durch einen kurzen Lo-Impuls an, daß Daten für den Drucker bereit stehen. Der Drucker akzeptiert die Daten und legt seinerseits die Leitung BUSY so lange auf Hi-Pegel, bis er das Zeichen verarbeitet hat und zur Annahme des nächsten Zeichens bereit ist. Bevor der Rechner das nächste Zeichen schicken kann, muß er erst abwarten, bis die BUSY-Leitung wieder auf Lo-Pegel ist. Zur Übertragung wird die CIA 2 des Commodore 64 benutzt. Port B, der User-Port, dient dabei zur Übertragung der Daten. Das Strobe-Signal geht über die Leitung PA2 (Bit 2 von Port A). Die Busy-Leitung des Druckers wird mit der Leitung FLAG des User-Ports verbunden. Bei einem Übergang von Hi- auf Lo-Pegel wird in der CIA automatisch das Bit 4 im Interrupt Control Register gesetzt. Wir können also damit genau erkennen, wenn der Drucker wieder bereit ist, Daten zu empfangen. Das folgende Timingdiagramm stellt diesen Sachverhalt noch einmal grafisch dar.



Doch jetzt zur Besprechung unseres Programms. Nach der Definition der Adressen finden wir zuerst die Initialisierung, die in üblicher Weise den Vektor für OPEN auf unsere neue Routine setzt. Die Routine selbst beginnt analog zur Betriebsystemroutine mit den Test der logischen Filenummer. Ist sie null, so geben wir eine Fehlermeldung aus. Ansonsten suchen wir nach einem geöffneten File mit dieser Nummer. Falls kein File mit der gleichen Nummer geöffnet war, können wir nun prüfen, ob bereits zehn Dateien offen sind. In diesem Fall ist die Kapazität der Filetabellen erschöpft und wir geben die Fehlermeldung 'too many files' aus. Jetzt können wir die Geräteadresse überprüfen. War es nicht vier, so springen wir in die normale OPEN-Routine zurück. Ansonsten erhöhen wir die Anzahl der offenen Files und tragen logische Filenummer, Geräteadresse und Sekundäradresse in die entsprechenden Tabellen ein. Die Pufferzeiger werden auf den Beginn des Puffers gesetzt. Als Puffer benutzen wir die 8 KByte unter dem Betriebssystem von $E000 bis $FFFF. Dann wird der User-Port auf Ausgang geschaltet und das Strobe-Signal auf Hi-Pegel gelegt. Jetzt werden noch die Vektoren für BSOUT und CLOSE auf unsere neuen Routinen gelegt. Das eigentliche Spooling geschieht im Interrupt; wir ändern den Interruptvektor dazu auf die Routine SPOOLING. Nachdem das Carryflag gelöscht ist, können wir mit RTS zurückkehren.

Die Spoolroutine, die in den Systeminterrupt eingebunden ist, schaltet zuerst die Speicherkonfiguration auf RAM um und prüft dann, ob ein Zeichen zur Ausgabe im Puffer steht. Dies ist dann der Fall, wenn der Schreibzeiger, der von der Routine BSOUT bei jedem Schreiben in den Puffer erhöht wird, nicht mit dem Ausgabezeiger übereinstimmt. Wenn der Drucker jetzt bereit ist, Zeichen anzunehmen, so holen wir ein Byte aus dem Puffer und legen es auf den User-Port. Durch

Umschalten der Strobe-Leitung auf Lo und anschließend wieder auf Hi zeigen wir dem Drucker an, daß ein Zeichen anliegt. Nun erhöhen wir den Lesezeiger, damit beim nächsten Male das folgende Zeichen aus dem Puffer ausgegeben werden kann.

Wir verzweigen jetzt zum Anfang der Routine und geben das nächste Zeichen aus. Die Schleife wird solange durchlaufen, bis entweder keine Zeichen mehr auszugeben sind oder bis der Drucker keine Zeichen mehr annimmt. Am Label EXIT wird dann die ursprüngliche Speicherkonfiguation wieder hergestellt und die normale Interruptroutine ausgeführt.

Die Routine BSOUTNEW testet lediglich, ob die Ausgabe auf Gerät 4 geht. In diesem Fall wird das Zeichen in den Puffer geschrieben und der Pufferzeiger erhöht. Die Routine zerstört keine Registerinhalte und wird mit gelöschten Carryflag verlassen, um anzuzeigen, daß keine Fehler aufgetreten sind.

In CLOSENEW werden, falls die Geräteadresse 4 erkannt wurde, die Vektoren für BSOUT und CLOSE wieder auf die ursprünglichen Werte gesetzt. Auch der Interruptvektor wird auf seinen alten Wert gesetzt. Dadurch wird die Ausgabe von evtl. noch im Puffer stehenden Zeichen abgebrochen. Soll dies vermieden werden, so müßte eine Warteschleife eingefügt werden, die so lange wartet, bis die Pufferzeiger für Schreiben und Lesen gleich sind.

Zum Anschluß des Druckers ist ein Kabel erforderlich, das den User-Port des Commodore 64 mit der Centronicsschnittstelle z.B. eines Epson-Druckers verbindet. Dabei sind folgende Leitungen zu verbinden:

USER-PORT	-	CENTRONICS
A	GND	16
B	FLAG - BUSY	11
C	D0	2
D	D1	3
E	D2	4
F	D3	5
H	D4	6
J	D5	7
K	D6	8
L	D7	9
M	PA2 - STROBE	1

Da die meisten Drucker mit Centronicsschnittstelle über den ASCII-Zeichensatz verfügen, der sich vom Zeichensatz des Commodore 64 unterscheidet, könnte bei der Ausgabe gleich noch eine Wandlung in den ASCII-Kode vorgenommen werden.

Bei der Inbetriebnahme ist folgendes zu beachten. Verbinden Sie Rechner und Drucker mit dem Kabel und schalten Sie dann den Rechner und danach den Drucker ein. Dadurch ist gewährleistet, daß der Drucker beim Übergang in den READY-Zustand das Flag-Bit in der CIA setzt. Jetzt können Sie das Maschinenprogramm laden und mit SYS 52224 initialisieren. Nach OPEN 1,4 werden mit PRINT# alle Daten in den Puffer im Rechner geschrieben, dessen Inhalt in der Interruptroutine an den Drucker geschickt wird. Das Schreiben in den Puffer geht dabei sehr schnell vor sich, so daß Ihr Anwendungsprogramm längst beendet sein kann, während die Interruptroutine noch damit beschäftigt ist, den Pufferinhalt zum Drucker zu schicken.

4.1 Tabelle der Schlüsselworte und ihrer Tokens

Token		Befehl	Adresse	Token		Befehl	Adresse
$80	128	END	$A831	$9F	159	OPEN	$E1BE
$81	129	FOR	$A742	$A0	160	CLOSE	$E1C7
$82	130	NEXT	$AD1E	$A1	161	GET	$AB7B
$83	131	DATA	$A8F8	$A2	162	NEW	$A642
$84	132	INPUT#	$ABA5	$A3	163	TAB(	-
$85	133	INPUT	$ABBF	$A4	164	TO	-
$86	134	DIM	$B081	$A5	165	FN	-
$87	135	READ	$AC06	$A6	166	SPC(	-
$88	136	LET	$A905	$A7	167	THEN	-
$89	137	GOTO	$A8A0	$A8	168	NOT	-
$8A	138	RUN	$A871	$A9	169	STEP	-
$8B	139	IF	$A928	$AA	170	+	$B86A
$8C	140	RESTORE	$A81D	$AB	171	-	$B853
$80	141	GOSUB	$A883	$AC	172	*	$8A2B
$8E	142	RETURN	$A8D2	$AD	173	/	$BB12
$8F	143	REM	$A93B	$AE	174	↑	$BF7B
$90	144	STOP	$A82F	$AF	175	AND	$AFE9
$91	145	ON	$A94B	$B0	176	OR	$AFE6
$92	146	WAIT	$B82D	$B1	177	>	-
$93	147	LOAD	$E168	$B2	178	=	-
$94	148	SAVE	$E156	$B3	179	<	-
$95	149	VERIFY	$E165	$B4	180	SGN	$BC39
$96	150	DEF	$B3B3	$B5	181	INT	$BCCC
$97	151	POKE	$B824	$B6	182	ABS	$BC58
$98	152	PRINT#	$AA80	$B7	183	USR	$0310
$99	153	PRINT	$AAA0	$B8	184	FRE	$B37D
$9A	154	CONT	$A69C	$B9	185	POS	$B39E
$9B	155	LIST	$A69C	$BA	186	SQR	$BF71
$9C	156	CLR	$A65E	$BB	187	RND	$E097
$90	157	CMD	$AA86	$BC	188	LOG	$B9EA
$9E	158	SYS	$E12A	$BD	189	EXP	$BFED

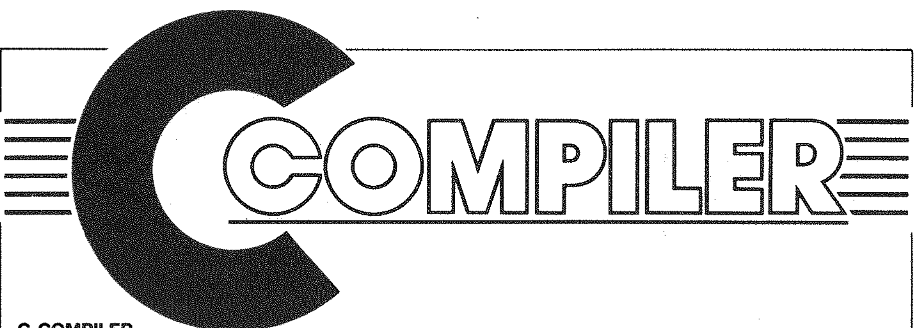
Token		Befehl	Adresse
$BE	190	COS	$E264
$BF	191	SIN	$E26B
$C0	192	TAN	$E2B4
$C1	193	ATN	$E30E
$C2	194	PEEK	$B80D
$C3	195	LEN	$B77C
$C4	196	STR$	$B465
$C5	197	VAL	$B7AD
$C6	198	ASC	$B78B
$C7	199	CHR$	$B6EC
$C8	200	LEFT$	$B700
$C9	201	RIGHT$	$B72C
$CA	202	MID$	$B737
$CB	203	GO	-

Die Tabelle ist so aufgebaut, daß zuerst die Befehlsworte kommen ($80 - $A2), dann folgen spezielle Worte, die im Zusammenhang mit anderen Befehle benutzt werden ($A3 - $A9). Daran schließen sich die Operatoren an ($AA - $B0). Nach den Vergleichsoperatoren ($B1 - $B3) kommen dann die BASIC-Funktionen ($B4 - $CA). Den Abschluß der Tabelle bildet der Kode für GO, der es erlaubt, GOTO auch als GO TO zu schreiben. Hinter den Befehlsworten steht, sofern dies möglich ist, die Adresse, an der die entsprechende Routine im ROM steht.

4.2 Zeropage-Vergleichstabelle C64 - C128

Bedeutung	C64	C128
Load/Verify-Flag	$0A	$0C
Typflag	$0D	$0F
Flag für Integer	$0E	$10
BASIC-Programmstart	$2B/$2C	$2D/$2E
BASIC-Variablenanfang	$2D/$2E	$2F/$30
BASIC-Arrayanfang	$2F/$30	$31/$32
BASIC-Arrayende	$31/$32	$33/$34
BASIC-Stringbeginn	$33/$34	$35/$36
BASIC-RAM-Ende	$37/$38	$39/$3A
aktuelle Zeilennummer	$39/$3A	$3B/$3C
FAC#1	$61-$66	$63-$68
FAC#2	$69-$6E	$6A-$6F
CHRGET	$70	$380
CHRGOT	$76	$386
TXTPTR	$77/$78	$3D/$3E
SR bei SYS-Befehl	$30F	$05
AC bei SYS-Befehl	$30C	$06
XR bei SYS-Befehl	$30D	$07
YR bei SYS-Befehl	$30E	$08
USR-Vektor	$311/$312	$1219/$121A

Wie Sie aus der Tabelle erkennen können, ergibt sich der zum 128er gehörende Wert aus dem 64er-Wert durch Addition von zwei. Dies gilt für die Zeiger des BASIc-Interpreters bis zur CHRGET-Routine.



C-COMPILER
DM 298,-*

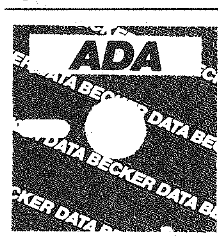


"C" ist die kommende Programmiersprache! Bereits heute arbeiten große Softwarehäuser mit ihr - sogar das bekannte Betriebssystem "UNIX" wurde in "C" geschrieben. Jetzt kann auch der C-64-Anwender diese zukunftsweisende Computersprache nutzen, die bisher nur den "Großen" vorbehalten war. Und zwar in vollem Umfang, denn der C-Compiler von DATA BECKER ermöglicht nicht nur einen Einblick in dieses hochinteressante System, sondern bietet eine Möglichkeit zur professionellen Programmerstellung - eine echte Alternative zu anderen Sprachen. Das C-Paket enthält Editor, Compiler, Dienstprogramme sowie ein ausführlich dokumentiertes Handbuch.

Der C-COMPILER in Stichworten:
EDITOR: Full Screen Editor mit variabler Zeichenbreite, maximal 43 K Textspeicher, 2 Zeichensätzen und komfortablen Textverarbeitungsfunktionen.
COMPILER: Voller Sprachumfang nach Kernigan und Ritchie (außer Bitfeldern), erzeugt direkten Maschinencode, hat 50 KByte Speicher für den objektcode verfügbar, optimiert Ausdrücke und bietet 16(!)-stellige Fließkomma-Arithmetik.
LINKER: Bindet bis zu 7 getrennt compilierte Quellfiles zu einem lauffähigen Maschinenprogramm zusammen, das sowohl vom C-Hauptmenue als auch von BASIC aus gestartet werden kann.
COPY: Diskettendienstprogramm.
Mit dem DATA BECKER "C"-COMPILER steht dem C-64-Anwender jetzt ein mächtiges Softwarewerkzeug zur Verfügung. "C"-Programme lassen sich ohne großen Aufwand auf PC's oder selbst auf Großrechner übertragen. Der DATA BECKER "C"-COMPILER wurde komplett in Deutschland von deutschen Autoren entwickelt. Natürlich mit ausführlichem Handbuch. Programm wird auf Diskette (für Diskettenlaufwerk 1541) geliefert.

* unverbindliche Preisempfehlung. Alle Programme auf Diskette für VC-1541.

ADA-Trainingskurs
DM 198,-*



Diese Programmiersprache der Zukunft, wie seinerzeit COBOL vom Pentagon in Auftrag gegeben, kann jetzt mit dem DATA BECKER-Trainingskurs auch der C-64-Anwender erlernen. Der ADA-Trainingskurs enthält außerdem einen Compiler, der einen umfassenden SUBSET und die wesentlichen Elemente dieser Sprache bietet.

ADA-Trainingskurs in Stichworten:
blockstrukturierte Programme - modularer Aufbau der Programme - ermöglicht die Behandlung von Ausnahmezuständen - lexikalische, syntaktische und semantische Fehlerüberprüfung beim Übersetzen und zur Laufzeit - Nennung der fehlerhaften Zeilennummer führt zu problemloser Fehlersuche - ermöglicht das einfache Einbinden von Maschinenprogrammen - ausgesprochen leichtes Arbeiten mit Programmbibliotheken - Konstanten und Variablendefinitionen von verschiedenen Typen (Integer, Strings etc.) - Assembler erlaubt Kommentare, Benutzung von Labels, verschiedene Zahlenformate, Pseudo-Anweisungen (z.B. ".BYTE", ".MARKE", ".START", ".BLOCK", ".WORD", ".COUNT" etc.) und die Verwendung aller Mnemonics (MOS Standard) - Disassembler ermöglicht die Analyse von Maschinenprogrammen - 62 Schlüsselwörter, trotzdem genug Speicherplatz für eigene Programme vorhanden - Programmdiskette enthält Editor, Übersetzer, Assembler und Disassembler - umfangreiches deutsches Handbuch mit Übungen und Lösungsvorschlägen zu den Themen Textausgabe, Bildschirmsteuerung, Datenobjekte, Datenein- und -ausgabe, Wertzuweisung, Entscheidungen und Schleifen - ausführliche Darstellung und Erläuterung der Grammatikregeln - Grammatikindex.

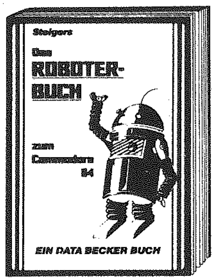
* unverbindliche Preisempfehlung. Alle Programme auf Diskette für VC-1541.

Diese hochkarätige Einführung in die rechnerunterstützte Konstruktion liefert neben umfassenden Informationen reichlich Konstruktionsbeispiele mit etlichen Programmen. Konkret werden dreidimensionale Zeichnungen und deren Veränderung durch Zoomen, Duplizieren, Spiegeln etc. behandelt, Bausteinprinzip und Macros erklärt sowie darüber hinaus der Aufbau eines eigenen CAD-Systems erarbeitet. Ein brandaktuelles Buch der absoluten Spitzenklasse!



Heift
Einführung in CAD mit dem Commodore 64
302 Seiten, DM 49,-
ISBN 3-89011-067-3

Steigers
Das Roboterbuch zum Commodore 64
ca. 230 Seiten, DM 49,-
erscheint April 1985
ISBN 3-89011-86-X



STAR-TRECK im Wohnzimmer? Dieses packende Buch zeigt, wie man sich einen Roboter ohne großen finanziellen Aufwand selbst bauen kann und welche erstaunlichen Möglichkeiten der C 64 zur Programmierung und Steuerung bietet - anschaulich dargestellt mit vielen Abbildungen und etlichen Beispielen. Dazu ein spannender berblick über die historische Entwicklung des Roboters und eine umfassende Einführung in kybernetische Grundlagen. Unentbehrlich für jeden Roboterfan!

Voß
Einführung in die Künstliche Intelligenz
Mit vielen Programmen für den C 64
395 Seiten, DM 49,-
ISBN 3-89011-081-9



Zentrales Thema aktueller Diskussionen: die Künstliche Intelligenz (KI). Eine ausführliche und interessante Einführung in deren Theorie und Einsatzmöglichkeiten, vom historischen Abriß über die "denkenden" und "lebenden" Maschinen bis zu Anwendungsbeispielen mit Programmen für den Commodore 64. Expertensystem, Such- und Auskunftsprogramm oder selbstlernende Programme werden ebenso dargestellt wie Computer-Kunst oder Simulationen.



Sasse
Compiler - verstehen, anwenden, entwickeln
336 Seiten, DM 49,-
ISBN 3-89011-061-4

Zu den wichtigsten Arbeitsmitteln des Programmierers überhaupt gehören Compiler, deren Grundlagen, Funktionen und Einsatzweise in diesem Buch systematisch erklärt werden. Auch die Entwicklung eines eigenen Compilers, lexikalische, syntaktische und semantische Analyse sowie Codegenerierung sind ausführlich beschrieben. Mit vielen nützlichen Programmen, speziell zugeschnitten auf den Commodore 64 - Pflichtlektüre für jeden ernsthaften Programmierer!

Konkurrenzlos! Dieses Buch enthält nicht nur eine umfangreiche Programmsammlung, sondern ist zuglich qualifiziertes Standardwerk (inklusive Tips und Tricks!) für die anspruchsvolle wissenschaftliche Nutzung des C 64. Mit Sortier- und Mathematikprogramm, Statistik und weiteren interessanten Programmen für Chemie, Physik, Biologie und Elektronik wird der 64er zur wissenschaftlichen Hilfskraft. Ein breites Spektrum, gut und ausführlich dokumentiert.



Severin
Commodore 64 für Technik und Wissenschaft
296 Seiten, DM 49,-
ISBN 3-89011-021-5

Ein unentbehrliches Arbeitsinstrument für den Commodore-64-Anwender! Fachwissen von A-Z bei allen Fragen zur Computerei im allgemeinen und zum 64er im besonderen. Gleichzeitig ein Fachwörterbuch, natürlich mit deutscher Erklärung der englischen Fachbegriffe. Insgesamt eine unglaubliche Vielfalt an Informationen, die grundsätzliches Verständnis ebenso fördern wie fortgeschrittene Programmierung.



Jordan/Schellenberger
Das DATA BECKER Lexikon zum Commodore 64
354 Seiten, DM 49,-
ISBN 3-89011-0013-4



Angerhausen/Brückmann/Englisch/Gerits
64 Intern
Das große Buch zum Commodore 64 mit dokumentiertem Schaltplan

Die Herausforderung für jeden ernsthaften Anwender! Alles über Technik, Betriebssystem und fortgeschrittene Programmierung des Commodore 64. Mit ausführlichem ROM-Listing, sorgfältig dokumentierten Originalschaltplänen zum Ausklappen, zahlreichen Abbildungen, Schaltbildern, Blockdiagrammen und - natürlich - mit anspruchsvollen Programmen. Mit diesem unentbehrlichen Buch lernen Sie Ihren C 64 erst richtig kennen.

352 Seiten, 2 Schaltpläne, DM 69,-
ISBN 3-89011-000-2



Brückmann/Gerits/Wiens
Das große Druckerbuch
369 Seiten, DM 49,-
ISBN 3-89011-020-7

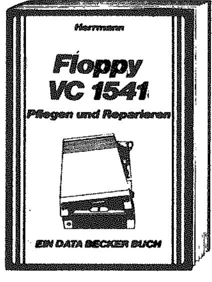
Mit diesem Buch meistern Sie absolut jedes Drucker-Problem. Ob Sekundäradressen, Schnittstellen, Steuerzeichen, formatierte Datenausgabe oder Grafik-Hardcopy: alles hervorragend erklärt. Selbstverständlich wieder viele nützliche Programme zum Abtippen; außerdem wichtige Hilfen zur Druckeranpassung, ein Betriebssystemlisting des MPS 801 und ein eigenes Kapitel zum VC-1520. Jetzt holen Sie das Optimum aus Ihrem Drucker heraus!

Das Standardwerk zur Floppy VC 1541. Alles über Diskettenprogrammierung vom Einsteiger bis zum Profi. Neben grundlegenden Informationen zum DOS, zu den Systembefehlen und Fehlermeldungen stehen mehrere Kapitel zur praktischen Dateiverwaltung mit der Floppy. Umfangreiches, dokumentiertes DOS-Listing. Dazu eine Fundgrube verschiedenster Programme und Hilfsroutinen, die das Buch für jeden Floppy-Anwender einfach zur Pflichtlektüre machen.



Englisch/Szczepanowski
Das große Floppy-Buch
482 Seiten, DM 49,-
ISBN 3-88011-006-3

Selbsthilfe spart Zeit, Ärger und Geld - gerade Probleme wie Floppy-Justage oder Reparaturen der Platine sind mit oft einfachen Mitteln zu lösen. Anleitungen zur Behebung der meisten Störfälle, Ersatzteillisten und eine Einführung in Mechanik und Elektronik des Laufwerks. Natürlich gehören auch genaue Angaben zu Werkzeug und Arbeitsmaterial zum Buch, das in jeder Beziehung für "effektiv und preiswert" steht.



Hermann
VC-1541 Pflegen und Reparieren
ca. 200 Seiten, DM 49,-
ISBN 3-89011-079-7

Das Superbuch, das Ihnen zeigt, was alles in Ihrem Rekorder steckt. Informiert detailliert und leichtverständlich über Datasette und Cassettenspeicherung. Mit den Spitzenprogrammen Autostart, Catalog (sucht und lädt automatisch!), Backup von und auf Floppy, Save von Speicherbereichen und einem neuen Cassetten-Betriebssystem mit dem 10-20 mal schnelleren (!) Fasttape. Außerdem weitere nützliche Hinweise (Kopfjustage, Kontroll-Lautsprecher) und Programme.



Paulissen
Das Cassettenbuch zum Commodore 84 und VC-20
190 Seiten, DM 29,-
ISBN 3-89011-030-4

Brückmann
Der Commodore 64 und der Rest der Welt
229 Seiten, DM 49,-
ISBN 3-89011-015-0



Literatur speziell für den engagierten Hobbyelektroniker vom fähigen Techniker zusammengestellt. Schwerpunkt sind ausgesuchte Ideen zu verschiedenen Einsatzmöglichkeiten des C 64: Motorsteuerung, A/D-Wandler, Spannungs- und Temperaturmessung und Lichtorgel. Dazu eine Reihe hochinteressanter Schaltungen zum Nachbau: EPROM-Programmer, Sprachsynthesizer, Frequenzzähler und noch mehr.